VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Generování kódu v e-PFL z UML diagramů

Generation of e-PFL Codes from UML Diagrams

2012 Juraj Vaněk

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Juraj Vaněk**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Generování kódu v e-PFL z UML diagramů
Generation of e-PFL Codes from UML Diagrams

Zásady pro vypracování:

Existují varianty jazyka UML určené k modelování vestavných systémů. Hlavním cílem práce je vytvoření nástroje, který by umožňoval generovat kód ve Vestavném procesně funkcionálním jazyce (Embedded Process Functional Language – e-PFL) z existujícího UML modelu. Jazyk e-PFL je hybridní funkcionální jazyk určený pro oblast vestavných systémů. Hlavní cíle práce lze shrnout v těchto bodech.

1. Zvolit vhodnou variantu UML. Jako vhodná varianta se jeví profil MARTE, který je poměrně rozšířený a má podporu v dostupných nástrojích. Také zvolte vhodný nástroj pro vytváření a zobrazování UML modelu a vhodný formát pro jeho uložení (například XMI).
2. Implementujte knihovnu, která umožní načtení a práci s vytvořeným UML modelem.
3. Vytvořte nástroj, který z UML modelu vygeneruje zdrojové kódy v jazyce e-PFL.
4. Řešení demonstrujte na ukázkovém příkladě.

K řešení použijte platformu .NET a programovací jazyk C#.

Seznam doporučené odborné literatury:

OMG MARTE:The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems, dostupné na: http://www.omg.org/spec/MARTE/1.0/PDF/.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Marek Běhálek, Ph.D.**

Datum zadání: 18.11.2011

Datum odevzdání: 04.05.2012

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

uhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 4. května 2012 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. května 2012 .....

Rád bych na tomto místě poděkoval svému vedoucímu diplomové práce Ing. Markovi Běhálkovi Ph.D. za pomoc při tvorbě této práce.

Abstrakt

Cílem této diplomové práce bylo vytvořit nástroj, který je schopen generovat kód hybridního funkcionálního jazyka e-PFL, určeného k modelování vestavných systémů z poskytnutých diagramů jazyka UML, popisujících daný systém. V teoretické části je obsaženo krátké představení jazyka e-PFL, dále krátký popis jazyka UML, u kterého byl kvůli obecné známosti kladen důraz spíše na ne příliš používané diagramy a také představení UML profilu MARTE. Tento profil rozšiřuje jazyk UML o možnosti modelování vestavných systémů. Dále je zde obsažena analýza problému transformace diagramů na kód, samotné implementace nástroje a dva ukázkové příklady.

Klíčová slova: UML, funkcionální jazyky, generování kódu, e-PFL, MARTE, vestavné systémy

Abstract

The purpose of this master thesis was the creation of a tool that is able to generate a code of a e-PFL hybrid functional language from provided UML diagrams. This language is designed to embedded systems modeling. The theoretical part contains a short introduction to e-PFL language, a brief description of the well known UML language with focus to newly added diagrams and introduction to MARTE profile. MARTE profile extends the UML language modeling capabilities for embedded systems. Second part consists of transformation analysis, description of tool implementation and two examples of tool use.

Keywords: UML, functional languages, code generation, e-PFL, MARTE, embedded systems

Seznam použitých zkratek a symbolů

UML	–	Unified Modeling Language
OMG	–	Object Management Group
XMI	–	XML Metadata Interchange
MARTE	–	Modeling and Analysis of Real Time and Embedded systems
e-PFL	–	Embedded Process Functional Language

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Vestavné systémy v současné době tvoří důležitou část našeho života a v budoucnosti jejich rozšíření dále poroste. Jako příklad mohou sloužit třeba gridové energetické sítě či chytré domácnosti, či obyčejné domácí spotřebiče. Ve velké části případů jde také o kritické systémy, jejichž výpadky mohou způsobit značné škody. Tyto systémy začínají také ve velké míře komunikovat mezi sebou a tím pádem nabývají na komplexnosti. Na druhou stranu zde vzniká tlak na co největší zkrácení času uvedení nových vestavných zařízení a systémů na trh.

Snížit riziko možného selhání systému a tím i náklady na případné pozdější opravy a úpravy (které mohou během plného nasazení daného systému růst do astronomických výšek) na úplné minimum lze důkladnou analýzou a návrhem požadovaného systému. Kvůli narůstající spletitosti systémů je analyzování a návrh systémů čím dál tím složitější a zejména u komplexnějších vestavných systémů s velkým množstvím paralelně pracujících prvků může být těžké nalézt možné kritické body. V tomto směru může být velmi nápomocný například jazyk *e-PFL*, určený pro modelování vestavných systémů, umožňující simulovat funkce a vzájemnou komunikaci mezi prvky modelovaného systému.

Účelem této práce bylo zamyslet se nad možnostmi generování kódu jazyka *e-PFL* z diagramů jazyka *UML*, který začíná být využíván při navrhování vestavných systémů a následně vytvořit nástroj, který toto generování umožní a který by mohl případně sloužit k urychlení procesu analýzy vestavného systému pomocí jazyka *e-PFL*. Vytvořené diagramy pak také mohou sloužit také při tvorbě dalších artefaktů procesu tvorby systému, či ke snadnějšímu pochopení kontextu, ve kterém došlo k případné chybě během simulace.

2 Jazyk e-PFL

2.1 Představení jazyka

Jazyk *e-PFL* (Embedded Process Functional Language - Vestavný procesně funkcionální jazyk) je hybridní funkcionální jazyk, který je určen pro modelování a analýzu vestavných systémů v raných fázích vývoje [1]. Slouží k tvorbě funkčního modelu vestavného systému či alespoň jeho nějaké části. Tento jazyk vznik z jazyka *PPFL* (Parallel Process Functional Language). Jazyk *PPFL* byl vytvořen kvůli potřebě tvorby souběžně pracujících procesů, určených pro simulaci paralelně spolupracujících funkčních jednotek vestavného systému. Samotný jazyk *PPFL* zde rozšiřuje jazyk *PFL* (Process Functional Language). Ten vychází z čistě funkcionální podmnožiny jazyka *Haskell* a tu obohacuje o možnost použití proměnných [1].

Jako funkcionální jazyk má jazyk *e-PFL* jednoduchou syntaxi, která je snadno pochopitelná a umožňuje rychlou tvorbu spustitelného návrhu vestavného systému. Umožňuje tak v raných částech vývoje systému ještě na vysoké úrovni abstrakce analyzovat spustitelný model vestavného systému a odhalit případné chyby a nedostatky.

2.2 Syntaxe

Jednotlivé prvky a pomocné funkce jsou definovány v základním modulu nazvaném *Prelude*.

Reprezentantem zařízení simulovaného vestavného systému je zde prvek nazvaný *Device*. Reprezentuje funkční jednotku daného systému. Definované jednotky *Device* jsou složeny z vestavných procesů. Tyto procesy jsou popsány datovým typem *EmbProcess*. Každý proces zde reprezentuje jednu funkci vykonávanou definovanou funkční jednotkou. Definice těchto procesů obsahují vstupní a výstupní proměnné, reprezentující komunikační kanály. Každá vstupní či výstupní proměnná procesu zde reprezentuje určitý vstup či výstup dané funkční jednotky. Stejně nazvaná proměnná u různých funkčních jednotek zde reprezentuje jeden společný komunikační kanál. Vestavný proces může být proveden pouze v případě, kdy jsou dostupné všechny vstupní proměnné, tzn. že jsou v odpovídajících komunikačních kanálech umístěny nějaké hodnoty. Na daném zařízení může být prováděn najednou pouze jeden proces. Proces lze provést pouze tehdy, pokud mají vstupní komunikační kanály daného procesu nastavenu nějakou hodnotu. Problém zde nastává, pokud je k provádění připraveno více procesů najednou. Proto musí být definován postup výběru určitého procesu k provedení. V jazyce *e-PFL* lze v tomto směru využít dva způsoby - *Fair* a *Unfair*. Tento přístup lze určit při definici dané funkční jednotky přímo v kódu (viz definice zařízení sensor ve výpisu kódu 1 níže).

Přístup typu Fair zde znamená, že jsou vybírány elementy na nižší úrovni tak, aby každý potomek ve stromě měl stejnou možnost být vybrán [1]. Přístup typu Unfair zase znamená výběr procesu podle pořadí v původní definici funkční jednotky [1].

Vstupní hodnoty, reprezentované komunikačními kanály ztotožněnými s parametry vestavného procesu jsou procesem při jeho provedení zkonzumovány [1]. Po provedení se daný proces snaží výsledek zapsat do komunikačních kanálů, které jsou reprezentovány

výstupními parametry. Pokud se to procesu nepodaří - tzn. že je daný komunikační kanál obsazen, není brán jako ukončený. Ukončeným se stane teprve, až se zápis do odpovídajícího kanálu podaří.

Funkční jednotky mohou být rozděleny do komponent. Ty jsou definovány pomocí datového typu *EmbSystem*. Tento typ má dva datové konstruktory. Prvním z nich je *Emulator*. Při použití tohoto konstruktoru nebude pro danou komponentu generován cílový kód, bude použita pouze při simulaci. Druhý konstruktor (*ESComponent*) má dva parametry. Pomocí prvního je definován název komponenty, druhý umožňuje definovat prostředníka. Prostředník zde v podstatě reprezentuje jazyk, ve kterém bude generován kód deklarované komponenty.

Vlastnosti spouštěných komponent lze ještě před spuštěním upravovat. Těchto úprav lze docílit pomocí anotací. Ty jsou definovány datovým typem *Annotation*. Programátorem použitelnou je zde například anotace *rename*, sloužící pro přejmenování některého ze vstupů či výstupů určité komponenty [1]. Touto anotací tak lze upravovat zapojení jednotlivých komponent a upravit tak výslednou strukturu simulovaného vestavného systému.

Po spuštění simulace již do struktury a funkčnosti jednotlivých komponent systému zasahovat nelze [1]. Spuštění celého systému je provedeno ve funkci *main*. Spuštění samotných komponent obstarává funkce *startDevice*. Ta přijímá tři vstupní parametry - zařízení, které bude daná komponenta reprezentovat (typ *Device*), následně komponentu (typ *EmbSystem*) a nakonec pole anotací, které spouštěnou komponentu před spuštěním upravují.

2.3 Ukázkový příklad použití syntaxe

```
import "Prelude.pfl"
...
sensorOutput :: a Integer -> ...
...
calibration :: ...
...
sensor :: Device
sensor = Unfair [Process sensorOutput, Process calibration]
...
sensorOne :: EmbSystem
sensorOne = ESComponent "sensor" Emulator
...
main = ... 'bl' (startDevice sensor sensorOne [rename "a" "c"]) ...
```

Výpis 1: Útržek kódu jazyka e-PFL

Použití syntaxe jazyka bude předvedeno na následujícím příkladu. Jedná se o simulaci systému, který obsahuje několik senzorů, řídící jednotku a jednotku simulující výstup pro uživatele.

V útržku kódu (výpis 1), reprezentujícího tento systém lze vidět import modulu *Prelude*. Pod ním se nachází definice procesů pojmenovaných jako *sensorOutput* a *calibration*. První z nich slouží k získání dat zachycených určitým senzorem, druhá definuje postup

při kalibraci senzoru. Následuje definice prvku *Device*, který je pojmenován jako *sensor*. Ten slouží jako určitý předpis jednotlivých komponent tvořících senzorovou síť. V definici můžeme vidět, že zde bude použit přístup *Unfair* a to, že *Device* zde bude obsahovat dvě výše definované funkce. Samotný jeden senzor je zde reprezentován komponentou pojmenovanou *sensorOne*. V jeho definici lze vidět název zařízení, kterou reprezentuje - *sensor*. Na konci se nachází hlavní metoda *main*, která má na starost korektní nastavení a spuštění simulovaného vestavného systému.

V tomto útržku kódu lze také vidět spuštění komponenty senzoru *sensorOne* pomocí funkce *startDevice*. V tomto konkrétním případě je zde také použita jedna anotace *rename*, která slouží k přejmenování komunikačního kanálu *a* na *c*.

3 Jazyk UML

Jazyk *UML* je určen k vizuálnímu modelování softwarových systémů. Lze jej použít jak pro pouhé znázornění detailů navrhovaného systému, zaznamenání myšlenek a návrhů vývojářů, tak i pro definování samotné architektury a návrhu systému, či pro následné generování kódu ve specifickém programovacím jazyce.

Tento jazyk vznikl na začátku devadesátých let spojením metodik vývoje softwaru vývojářů ze společnosti Rational Software, jmenovitě J. Rumbaugha, G. Booche a I. Jacobsona. Výsledkem byl návrh *UML* verze 0.9. V roce 1997 byl tento jazyk ve verzi 1.1. přijat standardizační organizací *OMG* (Object Management Group). V současné době již existuje verze 2.4.

Cílem konsorcia *OMG* bylo, jak již jeho název napovídá, specifikování a standardizace objektového přístupu programování. Aktuálně je spíše zaměřeno na modelování systémů, ale objektový přístup je promítnut i do tohoto jazyka.

Kvůli potřebě definování jazyka *UML* a vytvoření určitého nadhledu na něj vznikl standard *Meta-Object Facility*. Jedná se o architekturu umožňující definovat metamodely, ke kterým jazyk *UML* také patří. Tato architektura se skládá ze čtyř vrstev:

M3 - Nejvyšší vrstva - meta-metamodel. Umožňuje definovat samotné metamodely, umístěné o vrstvu níže.

M2 - Vrstva metamodelů, tvořících rámce pro modelování určitých problémů. Zde patří *UML*.

M1 - Vrstva uživatelem vytvořených modelů, například v *UML* vytvořený model vestavného systému.

M0 - Vrstva objektů reálného světa, popsaných pomocí modelu vrstvy M1.

K přenosu metadat, které lze vyjádřit pomocí *MOF*, byl vytvořen další standard - jazyk *XMI* (XML Metadata Interchange). Vychází ze značkovacího jazyka *XML*. Lze jej použít například k přenosu modelů vytvořených jazykem *UML*. Tento jazyk je v současné době využíván různými modelovacími nástroji, bohužel mnoho těchto nástrojů má tento standard přizpůsoben k vlastním potřebám.

3.1 Superstruktura jazyka UML

Superstrukturou tohoto jazyka zde označujeme všechny prvky jazyka *UML*, které lze použít na úrovni vrstvy M1 pro modelování potřebných systémů. Tento obecně známý jazyk umožňuje modelovat systém jak ze statického, tak dynamického pohledu. Současná verze obsahuje 14 druhů diagramů, 7 z nich popisující strukturu, dalších 7 určených pro popis chování. V následujícím textu bude kvůli známosti tohoto jazyka, hlavně základních částí, uveden pouze kratší přehled. Text bude spíše zaměřen na diagramy přidané v posledních verzích tohoto jazyka.

3.1.1 Strukturální diagramy

Diagramy jako třídní, komponentní či diagram nasazení v následujícím výčtu kvůli své obecné známosti rozebírány nebudou. Text bude zaměřen na ty méně používané.

Diagram objektů - V podstatě se jedná o specifický třídní diagram, znázorňující jednotlivé instance tříd a vztahy mezi nimi. Na rozdíl od třídního diagramu, který popisuje obecný pohled na statickou stránku projektu, tento diagram znázorňuje stav systému v jednom bodě času.

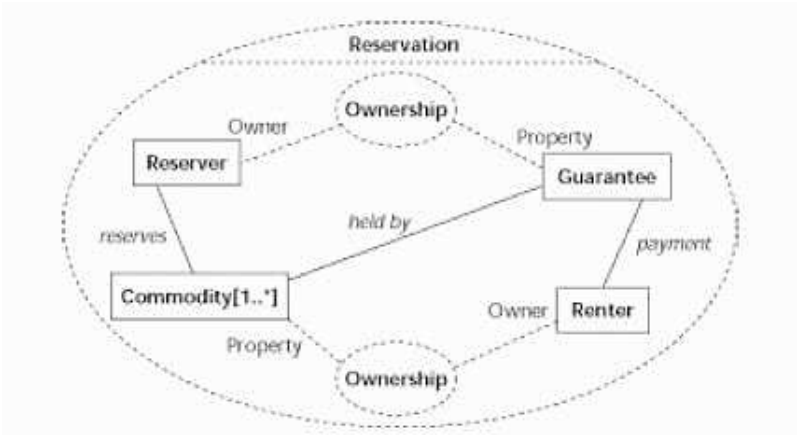
Diagram balíků - Pomocí tohoto diagramu lze znázornit rozdělení statických prvků systému do skupin a podskupin a jednotlivé vztahy mezi nimi. Jsou zde použity prvky z třídního diagramu, důraz je ale hlavně kladen na strukturu a vztahy mezi balíky.

Kompozitní diagram - Jedná se o diagram doplněný až ve verzi *UML 2.0*. Umožňuje definovat vnitřní strukturu klasifikátoru (objektů, které obsahují nějaké vlastnosti - proměnné a metody). Dovoluje popsat jeho vnitřní strukturu a vzájemnou interakci jeho jednotlivých součástí a interakci s okolním prostředím. Kompozitní diagram má několik způsobů použití. Lze jej použít pro tvorbu diagramu vnitřní struktury, který umožňuje znázornit vnitřní strukturu klasifikátoru. V něm jsou znázorněny jeho vnitřní části (jeho vlastnosti a objekty jiných tříd hrající určité role) a vztahy mezi nimi. Samotný klasifikátor zde tvoří obdélník s názvem klasifikátoru. Uvnitř tohoto obdélníku jsou umístěny jednotlivé role, tvořené opět obdélníky. Tyto role mohou obsahovat vlastní role a vlastnosti. Vlastnosti jsou v tomto diagramu zobrazeny jako obdélníky s okraji tvořenými přerušovanou čarou. Jednotlivé vztahy mezi částmi klasifikátoru jsou znázorněny pomocí plné čáry. Znázorněný klasifikátor může komunikovat s okolím pomocí portů (obdélníky umístěné na hraně obdélníku klasifikátoru). Tyto porty mohou využít i jednotlivé role obsažené uvnitř klasifikátoru.

Další možností využití kompozitního diagramu je tvorba diagramu spolupráce (Obrázek 1). Na rozdíl od předchozího diagramu vnitřní struktury jsou zde znázorněny jednotlivé instance klasifikátorů spolupracujících při plnění nějakého cíle systému. Ohraničení této spolupracující množiny instancí modelovaného systému je zde znázorněno pomocí oválu tvořeného přerušovanou čarou. Instance tříd zde opět tvoří obdélníky a jejich jednotlivé interakce jsou zde znázorněny plnou čarou.

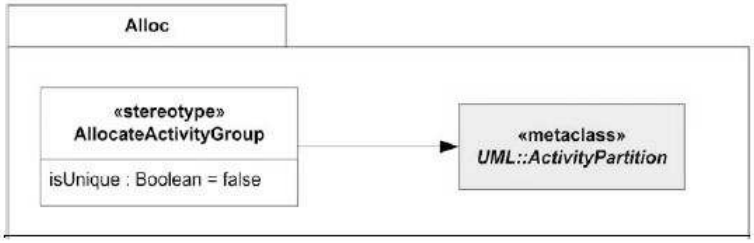
Na jednotlivé spolupráce instancí se lze odkazovat v diagramu vnitřní struktury. Tato reference je zde znázorněna oválem tvořeným přerušovanou čarou a názvem použité spolupráce uvnitř. Tuto referenci lze následně navázat na části klasifikátoru pomocí vztahu provázání rolí (*role binding*).

Diagram profilu - (Obrázek 2) Tímto diagramem lze vytvářet nové dialekty jazyka *UML*, které mohou lépe podchytit vlastnosti specifických systémů (jako například již zmíněný profil *MARTE*). Jedná se o klasický třídní diagram, ve kterém je umístěn balík



Obrázek 1: Ukázka diagramu spolupráce (Zdroj: http://orca.xf.cz).

označený pomocí stereotypu «*profile*» a ve kterém jsou definovány jednotlivé stereotypy profilu (třídy označené pomocí stereotypu «*stereotype*»). Tyto nově definované stereotypy lze propojit s prvky metamodelu jazyka *UML* (jako například *MetaClass* či *MetaAttribute*) pomocí vztahu *Extension* (v diagramu je znázorněn pomocí šipky reprezentované plnou čárou a vyplněnou hlavičkou). Tím jsou označeny prvky metamodelu, u na které lze daný stereotyp aplikovat. Samotné stereotypy lze také rozšiřovat jinými stereotypy. Pokud je stereotyp aplikován na nějaký prvek modelu systému, lze jeho vlastnosti použít jako tzv. *tagged values*, pomocí kterých můžeme daný prvek lépe popsat v rámci aplikovaného profilu.



Obrázek 2: Je zde vidět rozšíření prvku *ActivityPartition* metamodelu jazyka *UML* pomocí stereotypu *AllocateActivityGroup*. Balík *Alloc* zde není označen pomocí stereotypu «*profile*», protože se jedná o vnořený balík (*Zdroj: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*

3.1.2 Diagramy chování

Diagram případů užití - Tento diagram poskytuje funkční náhled na systém z pohledu uživatele a systémů interagujících s navrhovaným systémem. Je využit na počátku procesu vývoje systému a slouží pro sběr požadavků od budoucích uživatelů.

Diagram aktivit - Popisuje návaznost jednotlivých aktivit systému mezi sebou. K přechodu na další aktivitu může dojít pouze po skončení aktuální aktivity. Mezi hlavní prvky tohoto diagramu patří samotné aktivity, rozhodovací uzly a větve. Při návrhu systému se také podobně jako diagram případů užití používá spíše v dřívějších fázích vývoje.

Stavový diagram - Slouží k popisu možných stavů systému nebo jeho jednotlivých částí a možné přechody mezi nimi. Změnu stavu může vyvolat například změna hodnoty nějaké proměnné nebo volání nějaké metody.

Sekvenční diagram - Tento diagram slouží k popisu časových sekvencí interakce mezi jednotlivými objekty tříd systému. Patří mezi nejčastěji používané diagramy chování. Je užitečný hlavně při vizuálním znázornění nějaké důležitější interakce, při popisu složitějšího algoritmu interakce mezi objekty se může stát nepřehledným.

Komunikační diagram - Tento diagram interakce slouží ke znázornění specifické komunikace určité podmnožiny objektů tvořících systém. Lze jím popsat stejné situalce u kterých lze použít sekvenční diagram, nabízí ale jiný úhel pohledu.

Diagram přehledu interakcí - Pomocí tohoto diagramu lze modelovat tok řízení v rámci systému. Vychází z aktivitního diagramu, uzly aktivit jsou zde nahrazeny výskyty interakcí. Výskyt interakce je zde znázorněn pomocí klasického rámce sekvenčního diagramu, jenž odkazuje na specifický sekvenční diagram (interakční rámec *ref)*, nebo přímo znázorněným sekvenčním diagramem. Tyto uzly jsou propojeny toky aktivit, jejichž průběh je řízen uzly známými z aktivitního diagramu, jako je například rozhodovací uzel či větvení.

Diagram časování - Tento nový diagram jazyka *UML*, jenž je modifikací sekvenčního diagramu umožňuje znázornit časová omezení života modelovaného objektu systému. Lze jím zachytit změny stavu objektu v průběhu času. Horizontální osa diagramu zde reprezentuje čas (v podstatě se jedná o čáru života, používanou v sekvenčním diagramu a ve kterém je vedena vertikálně), a vertikální osa jednotlivé stavy daného objektu. Stejně jako u sekvenčního diagramu je zde znázorněn pouze jeden scénář, který reprezentuje určitý průřez funkčností systému.

4 UML při návrhu vestavných systémů

4.1 Současné možnosti modelování vestavných systémů pomocí jazyka UML

V současné době, kdy zvyšování výpočetního výkonu fyzických zařízení umožňuje vytvářet komplexní vestavné systémy za nízkou cenu a využívat jejich možností ve stále širší škále oblastí (automobily, energetika, domácí spotřebiče, atd.) roste i potřeba v těchto komplexních systémech nalézt co nejvíce chyb během raných fázích vývoje, kdy jsou náklady na opravu zjištěných chyb nízké.

V tomto směru je velice užitečná tvorba modelů vestavných systémů. Do těchto oblastí také začalo pronikat modelování pomocí jazyka *UML*. Mnoho postupů při vývoji software pomocí jazyka *UML* lze aplikovat i při vývoji vestavných systémů.

Jsou zde však oblasti modelování a analýzy těchto systémů, kde prostředky jazyka *UML* selhávají. Jedná se zejména o možnosti analýzy systému s ohledem na plánování provádění jednotlivých funkcí systému na různých, paralelně pracujících součástech a jejich vzájemné časování a synchornizaci (*schedulability analysis*) a o analýzu výkonu ([4]).

Při analýze a následném výběru softwarové a hardwarové platformy je také nutné brát v potaz omezení, které na klasický software být kladeny nemusí. Zde patří například pracovní podmínky, ve kterých musí spolehlivě fungovat či spotřeba elektrické energie samotného zařízení.

Kvůli potřebě zahrnout do modelování a analýzy systému tyto požadavky vzniklo několik rozšíření jazyka *UML*.

4.1.1 Dostupné metody modelování vestavných systémů pomocí jazyka UML

První z možných voleb jazyka pro vizuální modelování vestavných systémů je jazyk *SysML*. Jedná se o profil jazyka *UML* standardizovaný skupinou *OMG*. K vydání verze 1.0 došlo v listopadu 2005. Cílem tohoto profilu je využití jazyka *UML* při řešení problémů a modelování komplexních systémů v systémovém inženýrství. Obsahuje dva nové diagramy (diagram požadavků a parametrický diagram) a díky redukci částí jazyka *UML*, které jsou spíše zaměřené na software, je jeho syntaxe ve srovnání se syntaxí *UML* také jednodušší. Umožňuje modelovat širokou škálu systémů, například hardware, různá zařízení, informační a personální systémy atd.

V roce 2001 byl vytvořen profil nazvaný "UML Profile for Schedulability, Performance and Time", jehož účelem bylo lépe pokrýt modelování systému v oblastech vestavných systémů a popsat je lépe z pohledu časování, výkonu a interakce paralelně běžících částí těchto systémů.

Tento profil byl následně nahrazen profilem *MARTE* (Modeling and Analysis of Real-Time and Embedded systems). Ten by měl lépe pokrývat možnosti modelování vestavných systémů a jejich následnou analýzu.

5 UML profil MARTE

Toto rozšíření jazyka *UML 2* bylo vydáno v listopadu 2009 a momentálně je ve verzi 1.0. Profil je tvořen čtyřmi hlavními částmi, jejich podrobnější popis je uveden v následujícím textu.

5.1 MARTE Foundations

Jádro samotného profilu, obsahuje základní třídy, které jsou rozšiřovány v dalších částech. Hlavním balíkem této části je *CoreElements*. Ten obsahuje balíky *Foundations* a *Causality*. V balíku *Foundations* se nachází základní elementy pro popis samotných entit vestavného systému, balík *Casuality* zase obsahuje základní elementy pro popis chování modelovaného systému.

V balíku *CoreElements* se například nachází stereotyp *Configuration*, umožňující definování konfigurace systému, kterou reprezentuje množina nějakých aktivních elementů. Tento stereotyp rozšiřuje třídy metamodelu *UML StructuredClassifier* a *Package*. Dalšími stereotypy tohoto balíku lze rozšířit možnosti stavového diagramu. Stereotyp *Mode* zde umožňuje přidat další užitečné informace pro stav objektu, *ModeTranzition* k přechodům mezi jednotlivými stavy a *ModeBehavior* samotnému stavovému diagramu.

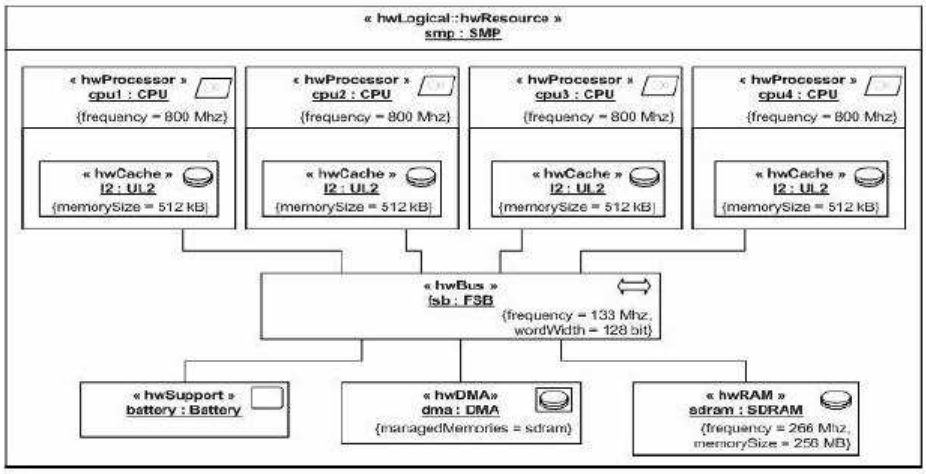
Dále se v této části nachází prvky umožňující popis nefunkčních vlastností vestavného systému, jako například taktovací frekvence, přenosové rychlosti, nebo vlastností *Quality of Service* (balík *NFPs - Non-Functional Properties*) a také stereotypy rozšiřující jazyk *UML* o možnosti modelování časových charakteristik vestavného systému (balík *Time*), které lze následně použít například ve výše uvedených nefunkčních vlastnostech systému.

Prvky, obsažené v balíku *NFPs* jsou například použity při definování různých veličin, obsažených v knihovně profilu *MARTE* (*MARTE Library*), umístěné v části *Annexes*. Je zde například umístěn výčet *PowerUnitKind*, definující jednotky výkonu (W, mW, kW). Samotný výčet je rozšířen pomocí stereotypu *Dimension*. V něm obsažené literály jsou zase rozšířeny pomocí stereotypu *Unit*.

Balík, zabývající se časovými charakteristikami systému v tomto profilu umožňuje například definovat jednotlivé hodiny, obsažené v modelovaném systému. Dokáže zde rozlišovat mezi chronometrickými a logickými hodinami (použitými např. při taktování procesoru). Dále umožňuje definovat například časem vyvolané události (stereotyp *TimedEvent*) nebo třeba rozšířit aktivity o informace kdy začínají, kdy končí či o jejich délce trvání.

Dalším balíkem, nacházejícím se v první části profilu *MARTE* je balík *GRM* (*Generic Resource Modeling*) umožňující na vysoké úrovni abstrakce modelovat části real-time systému bez ohledu na to, zda se jedná o software či hardware.

Základním prvkem tohoto diagramu je stereotyp *Resource* (rozšiřující prvky *Classifier*, *InstanceSpecification*, *Property*, *Lifeline* či *ConnectableElement*). Jedná se o abstrakci, umožňující definovat nějaký prostředek systému. Rozšiřují jej například stereotypy *ComputingResource* (např. procesor), *StorageResource* (např. paměť), *TimingResource* či *CommunicationMedia*.



Obrázek 3: Ukázka využití stereotypů balíku HRM při modelování hardware.

Dalším zajímavým stereotypem tohoto balíku je *Scheduler*. Umožňuje u rozšířeného elementu definovat přístup k množině prostředků poskytujících nějakou funkcionalitu (např. u *ProcessingResource*).

Poslední obsažený balík (*Alloc*) obsahuje elementy umožňující popis alokace funkčních elementů aplikace na dostupné zdroje. Mezi obsažené stereotypy zde patří *Allocated* (umožňuje určit elementy diagramu, které mohou být alokovány a také prvky, na které lze jiné alokovat), *Allocate* (rozšiřující vztah *Abstraction* a dovolující znázornit alokování prvků na poskytnuté prostředky), nebo například *Assign* (dovolujícící specifikovat samotnou alokaci pomocí komentáře).

5.2 MARTE Design Model

Tato část profilu rozšiřující *MARTE Foundations* se zabývá čistě návrhem samotného systému. Obsahuje tři hlavní balíky - *Generic Component Model* (*GCM*), *High-Level Application Modeling* (*HLAM*) a *Detailed Resource Modeling* (*DRM*).

Balík *GCM* obsahuje stereotypy, podporující komponentně orientovaný přístup k návrhu systému. Je zde vidět zaměření na prvky komponentního diagramu (např. stereotypy *FlowPort*, *ClientServerPort*). Zajímavým stereotypem je zde například *DataPool* (rozšiřující prvek *Property*), umožňující například definovat pomocí výčtu *DataPoolOrderingKind* typ řazení prvků (*LIFO*, *FIFO*, *UserDefined*), obsažených v dané vlastnosti. Několik zde obsažených stereotypů také rozšiřuje jednotlivé akce v aktivitním diagramu (*DataEvent*, *GCMTrigger*).

Balík *HLAM* zde rozšiřuje možnosti balíku *CoreElements*, umístěného v první části a balík *GRM*. Nejzajímavější jsou zde stereotypy *RtUnit* a *PpUnit* (oba rozšiřující třídu *BehavioredClassifier*). První z nich slouží k označení aktivní části systému, provádějící nějakou funkčnost, druhá zase umožňuje označit pasivní prvky systému (jako například

zdroje dat). U takto označených prvků systému lze také vybrat real-time služby, a to pomocí stereotypu *RtService* (rozšiřující *BehavioredFeature* - nadtřídu prvků *Operation* a *Reception*).

Posledním balíkem této části je *Detailed Resource Modeling* (*DRM*). Ten rozšiřuje balík *GRM* první části profilu a obsahuje dva hlavní balíky - *Software Resource Modeling* (*SRM*) a *Hardware Resource Modeling* (*HRM*). Jak jejich názvy napovídají, zabývají se již nižší úrovní návrhu a umožňují popsat systém detailněji jak z pohledu software, tak i hardware.

Balík *SRM* rozšiřuje velkou část stereotypů obsažených v balíku *GRM*. Obohacují je o vlastnosti, umožňující použít je při návrhu ze softwarového pohledu na modelovaný systém. Patří sem například stereotypy *SwTmerResource*, *SwSchedulableResource* (obě dědící ze *SwResource*) či *SwCommunicationResource* (dědící ze stereotypu *CommunicationMedia*).

Balík *HRM*, stejně jako balík *SRM* rozšiřuje mnoho stereotypů z balíku *GRM*, ale z pohledu hardware. Definované stereotypy umožňují do detailu definovat vlastnosti hardwarových částí. Například stereotyp *HwProcessor* (dědící z *HwComputingResource*), umožňuje u daného prvku definovat typ instrukční sady, počet jader, počet ALU a FPU jednotek atd.

5.3 MARTE Analysis Model

Třetí částí profilu *MARTE* je *MARTE Analysis Model*. Tato část se zabývá hlavně analýzou vestavného systému - umožňuje například specifikovat přístup paralelně pracujících částí ke sdíleným prostředkům, omezení týkající se výkonnosti, zabezpečení, využití paměti apod. Nacházejí se zde tři hlavní balíky - *Generic Quantitative Analysis Modeling* (*GQAM*), *Schedulability Analysis Modeling* (*SAM*) a *Performance Analysis Modeling* (*PAM*).

Balík *GQAM* definuje obecné koncepty pro různé typy analýz, a které jsou následně využity v následujících dvou balících.

Balík *SAM* umožňuje pomocí v něm obsažených stereotypů definovat parametry analýzy provádění jednotlivých funkcí systému a jejich časování. Část stereotypů zde rozšiřuje stereotypy balíku *GRM*. Umožňují zde například definovat minimální a maximální dobu trvání odeslání zprávy v sekvenčním diagramu.

Balík *PAM* zase pomocí svých stereotypů umožňuje analyzovat systém z pohledu požadavků na výkon, využití paměti apod.

Názorné příklady využití profilu *MARTE* při analýze vestavných systémů jsou uvedeny v materiálu *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems* [4].

5.4 Annexes

Poslední část profilu *MARTE* obsahuje přílohy doplňující předchozí části. Je zde například umístěn jazyk *VSL* (*Value Specification Language*) - textový jazyk pro specifikaci hodnot v rámci *UML* jazyka. Dále jsou zde obsaženy normativní knihovny, které rozšiřují primitivní typy, definované v jazyce *UML*, definují jednotlivé operace nad nimi a přidávající nový typ *DateTime*. Jsou zde také umístěny definice hodnot různých fyzikálních veličin (mm, m, J, W atd.), použitelných při modelování vestavných systémů.

6 Výběr modelovacího nástroje

Pro účely této práce je potřebné vybrat nástroj, ve kterém bude možno modelovat vestavné systémy a na jehož výstupu bude moci být provedena samotná transformace. V současné době existuje velké množství nástrojů umožňujících modelování systémů v jazyce *UML*. Nástroj potřebný pro další postup ale musí splňovat následující kritéria:

- Vytvořené modely musí být uloženy ve vhodném formátu dat. Ten musí být jednoduše zpracovatelný.
- Nástroj musí dovolovat modelování ve verzi 2.x jazyka *UML*.
- S předchozím požadavkem také úzce souvisí možnost rozšíření daného nástroje o využití profilu *MARTE*, který je rozšířením verze *UML 2.x*.

Z množiny dostupných nástrojů zde tedy automaticky vypadávají ty, které při ukládání projektů využívají proprietární přístup. Nejvhodnějším způsobem reprezentace dat je zde formát *XMI*. Druhý bod zde v současné době splňuje velká část nástrojů. Co se týče poslední podmínky, ta je splněna pouze u pěti následujících nástrojů:

IBM Rational Rhapsody - Prostředí pro modelování vestavných systémů a problémů systémového inženýrství. Podporuje generování kódu v jazycích jako je například Java, C, C++ či C#.

IBM Rational Software Architect - Modelovací a vývojové prostředí pro navrhování architektury v C++ a pro platformu J2EE. Je postaven na prostředí Eclipse.

MagicDraw - Komerční nástroj s podporou jazyků UML, BPMN či SysML a důrazem na spolupráci v týmu.

Modelio - Open-ource nástroj podporující UML2 a BPMN2. Rozšiřitelný o profil MARTE pomocí přídavného modulu.

Papyrus UML - Open-source nástroj postavený na prostředí Eclipse. Je také dostupný jako plugin pro zmíněné prostředí.

Z těchto nástrojů jsem následně zvažoval poslední dvě možnosti. Jedná se totiž o open-source nástroje, jsou tedy lehce dostupné. Nakonec jsem vybral nástroj *Papyrus UML*.

7 Papyrus UML

Jedná se o open source nástroj založený na prostředí *Eclipse*, určený pro modelování systémů pomocí jazyka *UML*. Lze jej rozšířit o možnosti modelování pomocí jazyka *SysML* (*System Modeling Language*), o *UML* profil *MARTE*, nebo například *CCM* (*CORBA Component Model*).

Uživatelské rozhraní má klasické rozložení nástroje *Eclipse* (viz obrázek 4) - vlevo je seznam dostupných projektů, umístěných v pracovní složce tohoto nástroje. Pod nimi je okno obsahující stromovou strukturu jednotlivých objektů *UML* modelu, obsažených v *XML* souboru projektu. V hlavním okně nástroje lze tvořit samotné diagramy pomocí prvků, které jsou umístěny v paletě úplně vpravo. Pod hlavním oknem je okno vlastností, kde může uživatel upravovat vlastnosti samotných *UML* objektů.

Ve výchozím stavu není žádný rozšiřující profil dostupný, profil *MARTE* je nutné doinstalovat pomocí klasického mechanismu instalace rozšíření platformy *Eclipse*. Podrobný postup je uveden na stránkách nástroje.

Při vytvoření projektu nového modelu je automaticky vytvořen diagram nazvaný *Default*. Ten umožňuje modelovat systém pomocí prvků dostupných v třídním a komponentním diagramu. Dále lze ze strukturálních diagramů jazyka *UML* přidávat třídní a komponentní diagramy, diagramy nasazení, diagramy případů užití či kompozitní diagramy. Zbývající diagramy struktury (objektový diagram, diagram balíků či diagram profilu) lze vytvářet pomocí třídního diagramu. Samostatně zde lze ještě přidávat sekvenční diagramy. Aktivitní a stavové diagramy samostatně vkládat nelze, mohou být vloženy pouze do již existujících strukturálních prvků modelu jako je například *Component* či *Class*. Chybí zde implementace použití diagramu časování a diagram přehledu interakcí.

Papyrus UML mimo jiné dovoluje reprezentovat jeden objekt systému v různých diagramech. Stačí pouze daný objekt vytvořit například ve třídním diagramu, a poté jej ze stromu struktury modelu zobrazeném v okně *Outline* přetáhnout do jiného diagramu.

7.1 Struktura uložených dat

UML model daného vestavného systému je v nástroji *Papyrus UML* uložen ve dvou souborech typu *XML*. První z nich s příponou **.di2* obsahuje grafickou reprezentaci jednotlivých diagramů a jejich prvků, druhý s příponou **.uml* reprezentuje jednotlivé prvky *UML* modelu a jejich vlastnosti. Struktura prvního souboru vypadá následovně:

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi= ... >
<di2:Diagram isVisible="true" fontFamily="Arial" lineStyle="solid" fontColor= ... name="
    DefaultDiagram">
  <contained xsi:type="di2:GraphNode" isVisible="true" fontFamily="Arial" ... size="723:345">
  ...
      <contained xsi:type="di2:GraphNode" isVisible= ... >
  ...
                    <semanticModel xsi:type="di2:Uml1SemanticModelBridge" ... >
                <element xsi:type="uml:DataType" href="myUMLModel.uml#
                    _zaZo8MXBEeCvJv0xhw8fDw"/>
```

```
                    </semanticModel>
                </contained>
            </contained>
            ...
            <contained xsi:type="di2:GraphEdge" isVisible="true" . . . anchor="/0/@contained.0/
                @anchorage.0␣/0/@contained.1/@anchorage.0">
            ...
            <owner xsi:type="di2:Uml1SemanticModelBridge">
              <element xsi:type="uml:Model" href="myUMLModel.uml#_pOLskMXBEeCvJv0xhw8fDw"/>
            </owner>
        </di2:Diagram>
        <di2:Diagram isVisible="true"  ...  type="ActivityDiagram">
            ...
```

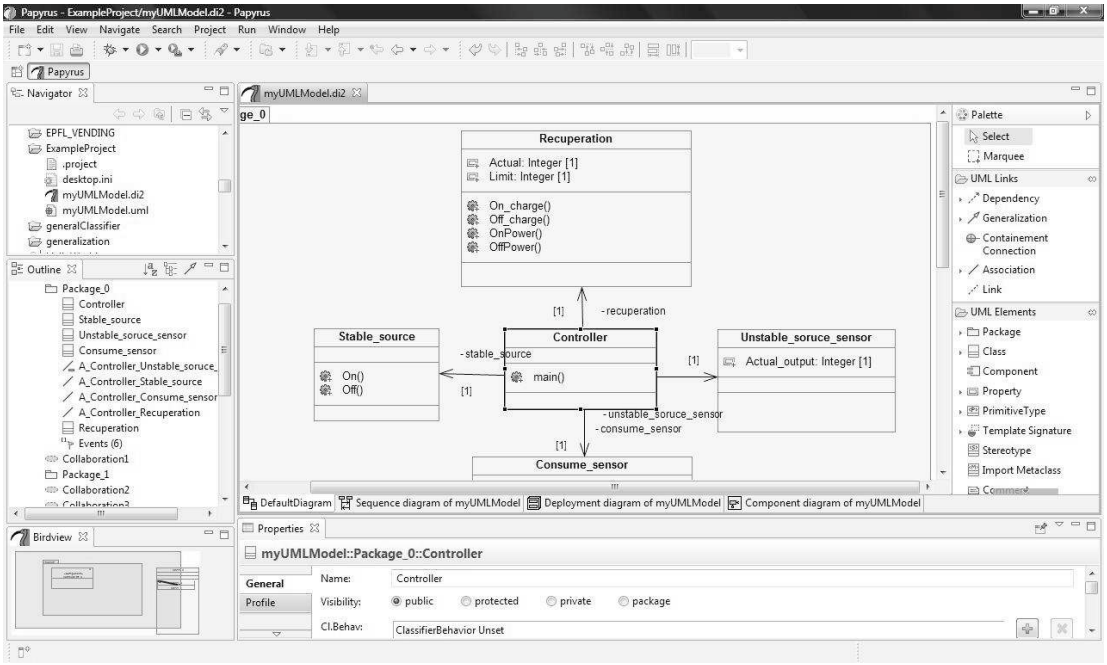
Výpis 2: Struktura XMI souboru diagramů

Je zde vidět jasné rozdělení na jednotlivé diagramy, které jsou umístěny uvnitř elementů *di2:Diagram* a struktura diagramů, skládajících se z uzlů (elementy *contained* s atributem *xsi:type*, který obsahuje hodnotu *di2:GraphNode*) a hran (opět elementy *contained*, ale s atributem *xsi:type*, obsahujícím hodnotu *di2:GraphEdge*). Jsou zde také vidět atributy obsahující informace o samotném zobrazení. V uzlech contained je také obsažen element *semanticModel*, který v hodnotě atributu *xsi:type* nese informaci o typu *UML* objektu a ve kterém je v atributu *href* umístěna reference na reprezentovaný *UML* objekt, obsaženém ve druhém souboru modelu. Struktura tohoto souboru vypadá následovně:

```
<?xml version="1.0" encoding="UTF−8"?>
<xmi:XMI xmi:version="2.1" xmlns:xmi= ... >
<uml:Model xmi:id="..."  name="myUMLModel">
<packageImport xmi:id="...">
<importedPackage xmi:type="uml:Model" href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.
    library.uml#_0"/>
</packageImport>
        <packagedElement xmi:type="uml:Class" xmi:id="␣..." name="Class_0" classifierBehavior="
            ...">
              <generalization xmi:id=" ... "  general=" ... "/>
            <ownedAttribute xmi:id="..."  name="Property_0" ... isUnique="false"/>
              <ownedBehavior xmi:type="uml:StateMachine" xmi:id="..." name="StateMachine_0"
                  >
                <region xmi:id=" ... "  name="Region_0">
    ...
    <packagedElement xmi:type="uml:SendSignalEvent" xmi:id="..." name="SendEvt1"/>
    <packagedElement xmi:type="uml:ReceiveSignalEvent" xmi:id="..." name="RecvEvt1"/>
    . . .
</uml:Model>
  <GRM:ComputingResource xmi:id="..." isProtected="true" isActive="true" base_Classifier="..."
      resMult="212"/>
  <Time:ClockType xmi:id="..." nature="dense" isLogical="true" base_Class="..."/>
    ...
```

Výpis 3: Struktura XMI souboru jazyka UML

Element *uml:model* zde reprezentuje samotný model vestavného systému. Uvnitř něj jsou zanořeny prvky *UML* jazyka, jako například zde element *packagedElement* s atribu-



Obrázek 4: Uživatelské rozhraní nástroje Papyrus UML

tem *xmi:type* obsahujícím hodnotu *uml:Class*, definujícím nějakou třídu. Je v něm vnořen element *ownedAttribute* určující vlastnost této třídy nazvanou *Property_0*. Dále je zde vidět, že tato třída dědí z jiné (element *generalization*). Je zde také definováno chování této třídy pomocí stavového diagramu (vnořený element *ownedBehavior*). Na konci souboru jsou umístěny jednotlivé události chování systému. Každý prvek modelu zde má určen jedinečný identifikátor umístěný v atributu *xmi:id*. V tomto souboru se také nacházejí prvky, které nejsou zobrazeny v žádném diagramu (tzn. že na ně není odkazováno z výše představeného souboru diagramů).

Při aplikaci rozšiřujícího profilu jazyka *UML* jsou aplikované stereotypy reprezentovány odpovídajícími elementy, umístěnými za element *uml:Model*. Dva takové případy lze vidět i v segmentu obsahu souboru **.uml* zobrazeném výše. Aplikované stereotypy profilu *MARTE* jsou zde reprezentovány elementy *GRM:ComputingResource* a *Time:ClockType*. Jakýkoliv element reprezentující nějaký stereotyp zde obsahuje stejně jako další elementy standardu *XMI* jedinečný identifikátor. V každém z nich musí být také obsažena informace o objektu, na který byl daný stereotyp aplikován. Ve výše uvedených případech jsou identifikátory odpovídajících objektů umístěny v atributech nazvaných *base_Classifier* respektive *base_Class*. Z těchto názvů jde mimochodem také vyčíst, jaký typ metamodelu daný stereotyp rozšiřuje. Zde se jedná o typ *Classifier*, který je nadřazenou třídou pro prvky jazyka *UML* jako je například *DataType*, *Interface* či *Class* a o typ *Class*, reprezentující obecně známý prvek třída.

8 Načítání UML modelu

8.1 Knihovna Loader

Pro načtení modelu systému do paměti ze souborů *XML* vytvořených v nástroji *Papyrus UML* byla vytvořena knihovna *Loader*. Při tvorbě této knihovny bylo zanedbáno načítání diagramů případů užití a prvky umožňující tvorbu nových profilů jazyka *UML*, protože z pohledu generování kódu jazyka *e-PFL* žádný smysl nemají.

Hlavní třídu této knihovny tvoří třída *Loader*, která obstarává načtení modelu z *XML* souborů a obsahuje reference na jednotlivé načtené diagramy. Pro využití této knihovny je zapotřebí nejprve nastavit názvy cest souborů struktury a diagramů, ze kterých má být struktura systému načtena. Pokud není nastaven název souboru diagramů, je automaticky hledán soubor ve stejné složce a se stejným názvem jako soubor struktury, ale s příponou změněnou na **.di2*. Ve třídě Loader jsou také obsaženy všechny načtené diagramy modelu. Každý typ diagramu zde reprezentuje jedna generická kolekce typu *List<Diagram>*.

Reference na všechny prvky modelu systému, které se nacházejí v diagramech jsou společně s referencemi na prvky, na které není ze souboru diagramů odkazováno (viz 7.1) umístěny do objetku typu *Pool* (viz 8.2), který obsahuje kolekce se všemi objekty obsaženými v načítaném *UML* modelu.

Pro samotné načtení struktury je potřeba zavolat metodu *LoadModel*, ve které jsou nejprve nastaveny jmenné prostory jazyka *XML*, které popisují elementy a atributy *XMI*, obsažené v načítaných souborech. Nejprve je načten soubor diagramů. Z něj jsou postupně vybrány elementy s názvem *di2:Diagram* a pro každý z nich zavolána metoda *AddDiagram*. Pomocí ní je zjištěn odpovídající typ diagramu a ze stromu elementů vnořených do elementu *di2:Diagram* jsou rekurzivním voláním privátní metody *GetIDs* získány identifikátory jednotlivých prvků, zobrazených ve vybraném diagramu. Ostatní informace nejsou potřebné, týkají se pouze formy zobrazení v pracovním prostředí nástroje.

U každého diagramu je také zjištěn jeho vlastník, objekty *UML* modelu zde totiž mohou vlastnit například různé diagramy chování. Pro každý získaný diagram je v metodě *AddDiagram* vytvořen odpovídající objekt a přidán do seznamu diagramů stejného typu. Následně je provedeno načtení objektů ze souboru s příponou **.uml*. V metodě *LoadModel* jsou vybrány pouze prvky na první úrovni *XML* stromu elementů, ty které jsou potomky kořenového elementu, případně potomky elementu *uml:Model*. Ty jsou poté vloženy do odpovídajících diagramů. Pro ověření, zda je v nějakém diagramu obsažen a k jeho případnému vložení do daného diagramu slouží metoda *DiagramOfElement*. Pokud není vybraný prvek obsažen v žádném diagramu, je vložen do výše zmíněného objektu typu *Pool*. Prvky zanořené na nižších úrovních stromu jsou načteny kaskádově a to nejprve během volání metod *Insert* odpovídajících tříd a poté v konstruktorech jednotlivých tříd, reprezentujících objekty *UML* diagramů.

8.2 Pool všech objektů modelu

Třída *Pool* zde reprezentuje entitu obsahující reference na všechny objekty diagramů jazyka *UML*, obsažených v načítaném modelu. Nachází se zde dvě kolekce typu *Dictionary<string,*

XMIObject>, nazvané *events* a *pool*. Klíč v těchto kolekcích tvoří identifikátor daného prvku modelu a hodnotu reference na něj. Kolekce *pool* zde obsahuje reference na všechny objekty modelu, kolekce *events* pouze reference na události jazyka *UML*. Ty nejsou nikdy v žádném diagramu zobrazeny, není tedy ani zapotřebí zjišťovat, zda je nějaký diagram obsahuje.

V této třídě se nachází dvě metody pojmenované jako *Insert*, kterými je možno do kolekcí vkládat reference. První z nich s parametry typu *XmlNode* a *Parameter* je zavolána v případě, že daný objekt není obsažen v žádném diagramu. V tomto případě je zjištěno, o jaký objekt se jedná a je vytvořena instance třídy, která jej reprezentuje.

Druhá metoda s parametry typu string a *XMIObject* slouží pro vložení reference již vytvořeného objektu.

8.3 Třídy reprezentující diagramy

Diagram *UML* modelu zde reprezentuje abstraktní třída *Diagram*. Ta obsahuje název daného diagramu, identifikátor objektu vlastníka tohoto diagramu a seznam identifikátorů prvků zobrazených v tomto diagramu. Dále tato třída implementuje rozhraní *IEquatable*, jehož implementovaná metoda *Equals* zajišťuje porovnávání různých instancí této třídy.

Důležitým prvkem této třídy je abstraktní metoda *Insert*, přijímající jako parametr objekt třídy *XmlNode*. Ta slouží k implementaci načítání prvků dané třídy, specifických pro jednotlivé diagramy.

8.4 Struktura kódu

Jednotlivé diagramy a jejich prvky jsou pro zpřehlednění rozděleny do složek. V každé z nich je umístěn soubor, reprezentující odpovídající třídu diagramu a další dvě složky - *elements* a *links*. První obsahuje prvky daného diagramu, druhá jejich vzájemné vztahy. Jedinou výjimku zde tvoří obsah složky *AcivityDiagram*, která obsahuje navíc složku *Actions*. Ta byla přidána pro zpřehlednění z důvodu vysokého počtu souborů obsahujících typy reprezentující jednotlivé akce aktivitního diagramu.

Každá třída diagramu obsahuje kromě implementace abstraktní metody *Insert* také kolekce jednotlivých prvků odpovídajícího diagramu.

Důležitým prvkem je v knihovně *Loader* také třída *Stereotype*, reprezentující aplikovaný stereotyp. Jsou v ní obsaženy vlastnosti, reprezentující název stereotypu, jmenný prostor, ze kterého stereotyp pochází, dále reference na *XML* element, který jej reprezentuje a nakonec samotný objekt stereotypu.

8.5 Zjednodušená reprezentace struktury jazyka UML

Samotný metamodel jazyka *UML* obsahuje více jak dvě stovky různých tříd, umožňujících popsat systém z různých úhlů pohledu. Jako příklad jeho obsáhlosti může sloužit soubor umístěný v příloze (*UML Superstructure.vsd*) vytvořený v aplikaci *Microsoft Visio*. Obsahuje část metamodelu jazyka *UML*, umožňující popis struktury systému a který jsem vytvořil kvůli tomu, abych získal lepší přehled o samotné struktuře tohoto jazyka popsané v

dokumentu *OMG Unified Modeling Language (OMG UML), Superstructure* [3]. Následně jsem jej také využil během úvah o samotné transformaci a výběru použitelných stereotypů profilu *MARTE*.

Reprezentace prvků jazyka *UML* je v souboru *XMI* rozdílná a přizpůsobená stromové struktuře jazyka *XML*.

Kvůli obsáhlosti struktury a rozdílné reprezentaci v souboru *XMI* jsem se rozhodl samotnou strukturu zjednodušit a přizpůsobit. Snažil jsem se vybrat ty nejpodstatnější části a strukturu tříd, reprezentující jednotlivé prvky jazyka *UML* přizpůsobit struktuře *XMI* a urychlit tak samotné načítání.

Při zjišťování struktury *XMI* jsem také vytvořil větší množství testovacích případů, které následně posloužily k otestování této knihovny.

Celková struktura knihovny *Loader* je vidět v třídním diagramu *ClassDiagram1*, umístěném ve složce projektu této knihovny. V textu níže budou popsány pouze její nejvýznamnější prvky.

8.5.1 XMIObject

Hlavní třídou, reprezentující prvek jazyka *UML* je zde třída *XMIObject*. Jedná se o třídu, ze které dědí všechny ostatní třídy, reprezentující prvky jazyka *UML*. Tato třída obsahuje dvě vlastnosti (*ID* a *IsInDiagram*) a kolekce diagramů, vlastněných instancí této třídy. *ID* zde reprezentuje jedinečný identifikátor v rámci načítaného modelu a je využit například v poolu všech prvků modelu (8.2). Vlastnost *IsInDiagram* zde pomáhá označit ty instance, které jsou v nějakém diagramu zobrazeny a má tedy smysl uvažovat je v následné transformaci. Ve struktuře modelu vestavného systému, obsažené v souboru *XMI* se mohou nacházet prvky, které byly v nějakém diagramu smazány a které nebyly již dále v jiném diagramu použity a nemusí mít tedy žádnou souvislost s modelovaným systémem. V této třídě jsou také obsaženy kolekce, obsahující diagramy, vlastněné tímto prvkem modelu. Nástroj *Papyrus UML* umožňuje vkládat diagramy přímo do určitého prvku jazyka *UML* a ten se následně stává jejich vlastníkem. *ID* vlastníka daného diagramu lze nalézt v souboru diagramů (viz struktura souborů 7.1). Pro práci s vlastněnými diagramy zde slouží tři metody - *GetOwnedDiagrams*, *DifferentOwner* a *GetOwnedBhaviorDiagram*.

Posledním důležitým prvkem této třídy je zde kolekce stereotypů, aplikovaných na tento objekt. Tu tvoří instance třídy *Stereotype*.

Z této třídy dědí například prvky jako *PackageImport*, *PackageMerge*, *Comment*, *Generalization* a další, u kterých není zapotřebí vlastností, obsažených ve třídách umístěných v hierarchii níže.

Další důležitou třídou, z dědící z této třídy je třída *Value*. Ta slouží jako rodičovská třída pro prvky *LiteralUnlimitedNatural*, *LiteralNull*, *OpaqueExpression*, *InstanceValue*, *LiteralString*, *LiteralBoolean* a *LiteralInteger*.

8.5.2 NamedElement, Element a ExtendedElement

Třída *NamedElement* rozšiřuje třídu *XMIObject* a umožňuje definovat název daného prvku. Tuto třídu dále rozšiřuje třída *Element*, obohacující prvky o možnost definice viditelnosti.

Z této třídy dědí například třída *Node* (a z něj dědící *ForkNode*, *JoinNode*, *DataStoreNode* atd.), třída *Message* (sloužící jako rodičovská třída zpráv, používaných v sekvenčním diagramu) či třída *Action* (reprezentující všechny možné akce, použitelné v aktivitním diagramu).

Dále z této třídy dědí například reprezentant vztahů mezi prvky třídního diagramu, třída *Link* (*Use* či *Dependency*) a jí rozšiřující *ExtendedLink* (mající pod sebou vztahy jako *Abstraction* či *Realization*).

ExtendedElement, dědící z prvku *Element* slouží jako rodičovská třída pro prvky *UML* jazyka jako *Activity*, *StateMachine* a *Interaction*. Dále tuto třídu rozšiřuje *DeploymentElement*, zaštiťující prvky diagramu nasazení (*Node*, *Device*, *Node* a *ExecutionEnviroment*).

Důležitým potomkem třídy *ExtendedElement* je také třída *Classifier*, obsahující operace a vlastnosti. Rozšiřují ji například třídy *Class*, *DataType*, *Signal* či *Interface*.

9 Načítání UML profilu MARTE

9.1 Knihovna MARTE

Stereotypy *UML* profilu *MARTE* jsou obsaženy v knihovně *dll*, nazvané *MARTE*. Během tvorby jsem se snažil zachovat strukturu, definovanou ve specifikaci. Kvůli rozsáhlosti profilu a potřebě pouze specifické množiny stereotypů (viz využití tohoto profilu během transformace 10.3) jsem neimplementoval vše.

I tak má tato knihovna aktuálně 142 tříd a pomocných výčtů. Některé z obsažených prvků využívají třídy obsažené v knihovně *Loader*.

9.2 Načítání stereotypů

Samotné načtení stereotypů, aplikovaných na jednotlivé prvky *UML* modelu má zde na starosti třída *MARTE_Loader*, která je obsažena v knihovně *MARTE*. V jejím konstruktoru dojde pomocí reflekce k načtení všech typů, obsažených v této knihovně. Ty jsou následně využity při načítání stereotypů. Použití reflekce také umožňuje případně strukturu této knihovny dále upravovat.

Nejprve jsou načteny jednotlivé jmenné prostory a jejich obsah. K jejich reprezentaci zde slouží třída *Namespace*, obsahující název jmenného prostoru a kolekci obsažených typů.

Názvy jednotlivých elementů souboru *XMI*, reprezentujících stereotypy se skládají ze dvou částí - z přímo nadřazeného jemnného prostoru a samotným názvem daného stereotypu, viz 7.1. Proto jsou při načítání obsahu knihovny brány v potaz pouze tyto části cesty k danému prvku v knihovně.

Pro samotné načtení stereotypů a přiřazení odpovídajícím částem modelu zde slouží metoda *LoadStereotypes*, která má dva vstupní parametry - kořenový uzel stromu *XMI* struktury modelu a referenci na *pool* objektů modelu, viz 8.2.

Jsou zde postupně procházeni potomci kořenového uzlu, a pokud odpovídají některému typu z knihovny *MARTE*, je zjištěn identifikátor objektu (viz 7.1), se kterým je tento stereotyp provázán, odpovídající objekt vybrán z poolu a poté odpovídající stereotyp vložen do kolekce stereotypů tohoto objektu.

10 Transformace

10.1 Pohled ze strany jazyka e-PFL

Pro pohled na strukturální stránku vestavného systému zde nejlépe poslouží třídní diagram. Jedná se totiž o ten nejzákladnější diagram a generovat kód z modelu, vytvořeném v jazyce *UML* a neobsahujícím žádný třídní diagram, nebo jakýkoliv diagram znázorňující strukturu nemá smysl. Mnoho prvků třídního diagramu je navíc použito i v ostatních diagramech jazyka *UML*.

10.1.1 Prvky Device

Základním prvkem jazyka *e-PFL* je prvek *Device*. Ten obsahuje nějaké procesy a pro jejich vykonávání může využívat další funkce a proměnné. V *UML* jazyce může tento prvek reprezentovat jakýkoliv typ, dědící ze třídy *Classifier*. Tato množina prvků zahrnuje prvky jako je *Class*, *Component*, *Signal*, *Interface*, *DataType* atd.

Device je v podstatě abstrakcí jednotlivých komponent. Pro další úvahy bude použit diagram na obrázku 5. V třídním diagramu může samotné *Device* reprezentovat prakticky jakýkoliv prvek. Jednotlivé prvky třídního diagramu zde mohou reprezentovat jak aktivní části - jednotlivé prvky *Device* a komponenty *EmbSystem* - tak i pasivní části - například zdroje dat.

Na extrakci potřebných reprezentantů zařízení ze strukturálního diagramu by se dalo dívat několika různými způsoby.

Vztah mezi *Device* a *EmbSystem* by zde mohl být například reprezentován pomocí nějakého vztahu abstrakce nebo generalizace. Při pohledu na diagram v obrázku 5 nastává problém, pokud bude *Device* reprezentován třídou *ClassA* a komponenty *EmbSystem* budou reprezentovány prvky, které z ní dědí. *ClassC* tvořící kvůli dědění ze třídy *ClassA* komponentu, také realizuje interface *InterfaceE*. Pokud bude i toto rozhraní považováno za reprezentaci *Device*, jakým způsobem bude ve vygenerovaném *e-PFL* kódu tento vztah reprezentován?

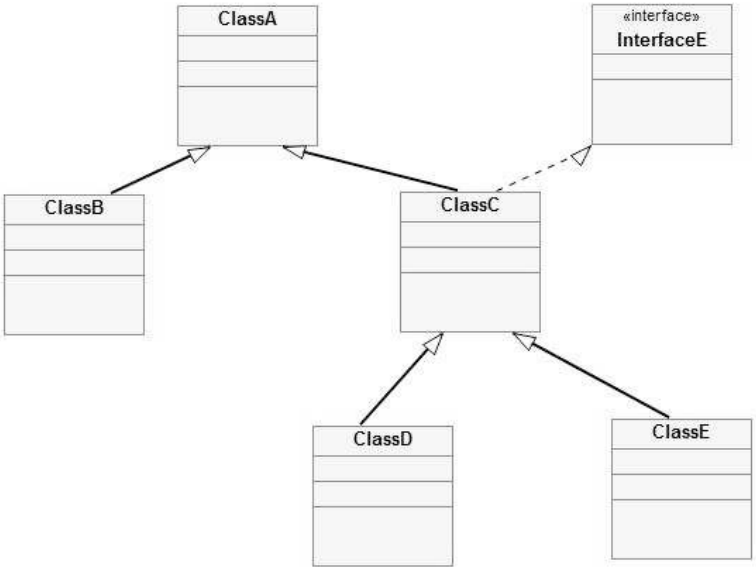
Lze také uvažovat případ, kdy aktivní komponentu tvoří jediná třída, nerozšiřující žádný další prvek diagramu. Prvek *UML* diagramu typu *Classifier* může tedy v jednom reprezentovat jak *Device*, tak i samotnou komponentu *EmbSystem*.

Ve výsledku tedy z třídního diagramu automaticky moc informací vyextrahovat nelze, bude zde zapotřebí pomoc uživatele, který vybere aktivní prvky modelovaného vestavného systému. Více informací půjde získat, pokud bude v modelu aplikován *UML* profil *MARTE*.

10.1.2 Prvky EmbSystem

Pro prvky *EmbSystem*, reprezentující konkrétní výskyt daného zařízení *Device* lze aplikovat stejnou úvahu jako pro samotný prvek *Device* popsanou výše. Bez pomoci uživatele se zde nástroj také dále nehne.

Pro co nejmenší zátěž uživatele během procesu transformace bude také výhodnější po něm požadovat pouze výběr komponent a případně ještě jejich množství. Uživatel



Obrázek 5: Třídní diagram

bude také muset definovat ty vlastnosti komponent, které nelze implicitně získat z *UML* diagramů.

Z uživatelem definovaných komponent lze pak automaticky zpětně definovat jednotlivá zařízení *Device*.

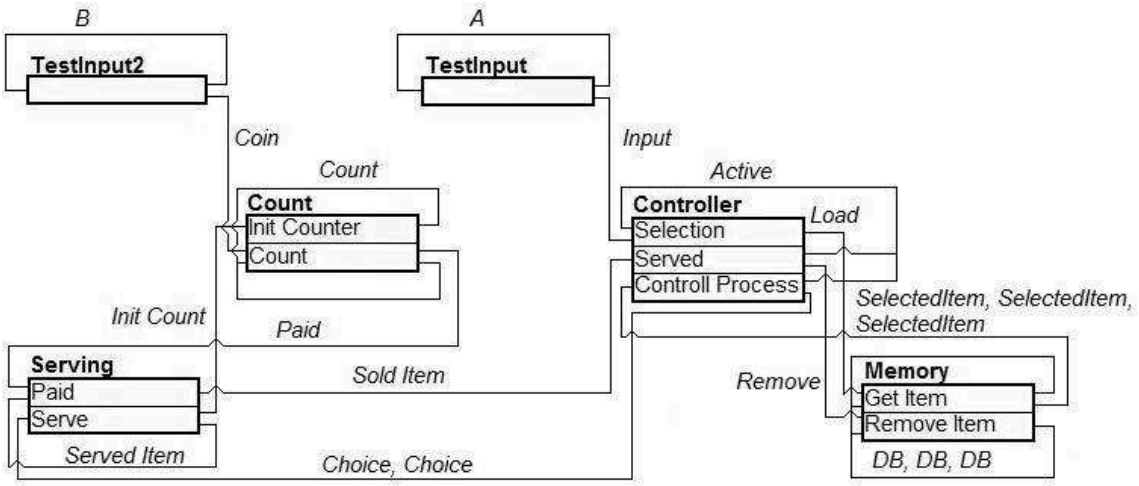
10.1.3 Procesy

Pro lepší znázornění úvah obsažených v následujícím textu jsem vytvořil obrázek 6. Jedná se o grafické znázornění kódu jazyka *e-PFL* popisujícího chování výdejního automatu. Jsou v něm názorně vidět jednotlivé komponenty systému, jejich procesy a jednotlivé komunikační kanály. Z původního kódu byly v obrázku kvůli zpřehlednění vynechány tři komponenty, mající na starost o výstup na obrazovku.

Lze v něm názorně vidět problémy, které mohou nastat při generování jazyka *e-PFL* z *UML* diagramů a které budou podrobněji rozvedeny níže.

Proces může v prvku *UML* diagramu typu *Classifier* (dále klasifikátor) reprezentovat jakákoliv obsažená operace či vlastnost. Je tedy zapotřebí nějakým způsobem zjistit, které to jsou. Některé z operací totiž mohou tvořit pomocné funkce daného prvku *Device*, volané v rámci procesu, ty v množině procesů zahrnuty nebudou.

V jazyce *e-PFL* je proces jednolitý kus kódu, který čeká na doplnění všech vstupních parametrů a během svého provádění s ostatními procesy nekomunikuje, ale až na konci vrátí výstupní parametry. Pomocí jazyka *UML* lze ale modelovat i situace, kdy je z jedné metody volána operace jiného prvku, který může pracovat paralelně a může tedy reprezentovat během simulace jiné zařízení. Tyto případy je třeba nějakým způsobem ošetřit. Jako příklad bude použita následující situace:



Obrázek 6: Výdejní automat

```
bool Operation (int a, int b)
{
    ...
    return !comp2.Call(a + b);
}
```

Výpis 4: Ukázková operace

Metoda *Operation* nějakého objektu zde pro provedení své funkcionality a vrácení požadovaného výsledku potřebuje výsledek metody *Call* objektu reprezentujícího jinou komponentu, zde nazvaného *comp2*. Pokud bude tato metoda považována za proces nějakého prvku *Device*, bude zapotřebí ještě před provedením tohoto procesu získat výsledek metody *Call*.

Je tedy třeba navíc do vstupních parametrů procesu umístit výstup této metody. Na to abychom tento výstup získali, musíme tomuto procesu poskytnout nějaká vstupní data. Ty jsou ale výsledkem provádění samotného procesu *Operation*.

Na první pohled by se mohlo zdát, že se tyto operace dostaly do stavu, kdy na sebe navzájem čekají. Tento problém lze ale vyřešit definováním počátečních hodnot pro parametry *a* a *b* ve funkci *main*. Výsledné úseky kódu jazyka *e-PFL* budou vypadat takto:

```
Operation :: c Bool -> a Integer -> b Integer -> (r Bool, in Integer)
Operation c a b = ...
    ...
Call :: x Integer -> y Bool
Call x = ...
    ...
setA :: a Integer -> ()
setA x = ()
    ...
setB :: b Integer -> ()
```

```
setB x = ()
    ...
main = setA 2 ´bl´ setB 3  ...
```

Výpis 5: Výsledná struktura

Účelem simulace vestavného systému pomocí jazyka *e-PFL* je také tyto případná uváznutí na mrtvém bodě nalézt a vyřešit je ještě během analýzy modelovaného vestavného systému.

Případ popsaný výše by mohl být například zobrazen podobně, jako proces *GetItem* zařízení *Memory*, znázorněném v obrázku 6. Tento proces také potřebuje pro získání vstupních proměnných, a tedy jeho provedení výsledek ze dvou různých procesů jiných zařízení.

Na obrázku zmíněném výše lze také například vidět případ procesu (proces *Served* zařízení *Controller*), mající za vstup parametr pocházející pouze z jednoho jiného procesu a jehož výstup je také použit pouze v jednom procesu. Tento případ lze v pseudokódu *OOP* jazyka by šel zapsat následujícím způsobem (pro zjednodušení byl další vstupní parametr procesu *RemoveItem* zařízení *Memory* odstraněn):

```
int Served (int soldItem)
{
   int remove = ...
    ...
   return RemoveItem(remove);
}
```

Výpis 6: Reprezentace procesu s jedním vstupním a jedním výstupním parametrem

V tomto případě není zapotřebí přidávat do vstupních či výstupních parametrů další dodatečné.

Také by se dalo uvažovat například nad případem, kdy by do procesu vstupovala data z jiného procesu a na jeho výstupu by závisely dva jiné procesy. Samotný pseudokód by mohl vypadat následovně:

```
int methodA (...)
{
   x[] = methodB(parameterA);
    ... methodC(x[0], parameterB);
    ... methodD(x[1]);
}
```

Výpis 7: Reprezentace procesu s jedním vstupním a dvěmi výstupními parametry

Jedná se o sekvenci příkazů, jejichž výsledkem je předání výsledků do dvou jiných procesů (či metod paralelně pracujících objektů). Sekvence příkazů v pseudokódu, reprezentujících tuto situaci ale může být také úplně jiná.

Tyto situace se vstupními a výstupními daty, pocházejícími ze dvou různých procesů lze zobecnit i na vyšší počty nezávislých procesů, z nichž jsou získávána vstupní data, případně jim zasílána výsledná dat.

Jde mi o to ukázat, že zde bude muset být také bráno v potaz případné přidávání parametrů do procesů modelovaného systému tak, aby šlo o vhodnou reprezentaci systému a to z důvodu nemožnosti interagovat s ostatními částmi systému během provádění samotného procesu.

Modelovaný systém bude s největší pravděpodobností odpovídat ukázkovému příkladu a může zde být problém automaticky určit vhodné přidané parametry. Záleží zde hodně na poskytnutých diagramech, s největší pravděpodobností nebude důkladně popsáno celé chování systému. Proto je zde důležité nejprve vytvořit scénář, který bude počítat pouze s třídním, případně jiným strukturálním diagramem, interakcí s uživatelem a samotný proces poté případně rozšířit o analýzu doplňkových diagramů. Diagramy chování v modelu vestavného systému budou spíše popisovat speciální scénáře než celkové chování systému. Stěžejním je zde vždy třídní diagram.

Provázání procesů vestavného systému proto musí být během samotné transformace brána v potaz a bude zde nutná také účast uživatele.

10.1.4 Komunikační kanály

Komunikační kanály mezi jednotlivými zařízeními ve vestavném systému simulovaném pomocí jazyka *e-PFL* jsou reprezentovány proměnnými, nacházejícími se na vstupech a výstupech jednotlivých procesů.

Každý komunikační kanál je zde reprezentován jedinečným názvem. Zde nastává problém při definici vestavného systému pomocí jazyka *UML*, ten totiž žádný takovýto mechanismus implicitně neobsahuje. Systém bude s největší pravděpodobností modelován bez ohledu na tento mechanismus. Diagramy chování nemusí kvůli zaměření spíše na specifické scénáře poskytnout dostatek vodítek pro automatickou konstrukci těchto komunikačních kanálů. Proto bude i zde nutná intervence uživatele.

Definice jednotlivých komunikačních kanálů a provázání jednotlivých procesů úzce souvisí. Název vstupního parametru jednoho procesu svázaného s výstupním parametrem musí odpovídat názvu tohoto výstupního parametru a reprezentovat tak jeden komunikační kanál. Proto by měl být proces provázání parametrů procesů obsažen společně s definicí komunikačních kanálů v jednom kroku transformace.

10.2 Pohled ze strany jazyka UML

Diagramy jazyka *UML*, reprezentující vestavný systém nemusí obsahovat kompletní informace o daném systému. Tyto informace také mohou být nepřesné a nekonzistentní a nelze jím obsáhnout některé vlastnosti. Následující text má za úkol popsat následnou reprezentaci co nejvíce částí jazyka *UML* ve výsledném kódu a to tak, aby byl co nejvíce zachován původní význam a funkčnost modelovaného systému, který bude pomocí jazyka *e-PFL* simulován.

Stereotypy profilu *MARTE* budou v textu rozebrány později. Uživatel je při modelování vestavného systému nemusí použít, proto zde bude reprezentován nejprve pohled při použití čistého jazyka *UML*.

Následující text je zaměřen na reprezentaci jednotlivých prvků jazyka *UML* při generování pomocného kódu. Samotné prvky jazyka *e-PFL* jako *Device*, procesy či *EmbSystem* budou vybrány přímo uživatelem, viz 10.1.2.

10.2.1 Strukturální pohled

10.2.1.1 Třídní diagram a prvky společné pro všechny diagramy Nástroj *Papyrus UML* při tvorbě nového projektu modelu vytvoří výchozí diagram (viz 7). Ten obsahuje prvky třídního diagramu, ale umožňuje vkládat i komponenty. Třídní diagram také obsahuje mnoho prvků využitelných v ostatních typech diagramů. V této části práce budou postupně rozebrány tyto společné prvky, od těch nejsnáze transformovatelných až po ty komplexnější. Následující text se bude zabývat prvky, které budou transformovány na pomocný kód.

10.2.1.1.1 Komentář, omezení a výčet První prvky, které zde budou rozebrány, jsou komentáře a omezení. Komentář zde jistě nemá smysl podrobněji rozebírat, v kódu *e-PFL* bude zobrazen jako klasický komentář (bude uveden pomocí "- -"). Omezení je v jazyce *UML* v podstatě text, obsahující omezující pravidla, kladená na daný objekt či systém, se kterým je toto omezení propojeno. Toto omezení může být napsáno jako obyčejná poznámka, nebo může být specifikováno pomocí speciálního jazyka, jako je například *OCL* (*Object Constraint Language*).

Pokud by bylo omezení popsáno například pomocí výše zmíněného *OCL*, mělo by smysl se jím hlouběji zabývat, pro začátek ale bude ve výsledném kódu reprezentován jako prostý komentář. Pro odlišení od klasického komentáře bude text omezení uveden pomocí komentáře " - - *CONSTRAINT* > ".

Oba tyto prvky budou uvedeny u každého fragmentu kódu reprezentujícího objekt *UML* modelu, se kterým jsou propojeny.

Dalším, jednoduše reprezentovatelným prvkem jazyka *UML* je výčet. V jazyce *Haskell* má svého přímého reprezentanta [2], v generovaném kódu jazyka *e-PFL* bude ale reprezentován jako datový typ s odpovídajícím názvem. Jednotlivé literály zde budou reprezentovat datové konstruktory tohoto datového typu.

Reprezentaci výčtu v jazyce *e-PFL* lze vidět ve fragmentu kódu níže, společně s komentářem a omezením.

```
    ...
−− CONSTRAINT > Text omezení kladeného na výčet
−− COMMENT > Text komentáře propojeného s výčtem
data Colors = Black | White
    ...
```

Výpis 8: Prvky jazyka UML v e-PFL kódu

10.2.1.1.2 Primitivní typy a viditelnost Jazyk *UML* obsahuje několik předdefinovaných primitivních typů - *Boolean*, *Integer*, *UnlimitedNatural* a *String*. Všechny z nich mají své protějšky i v jazyce *e-PFL*. Typu *Boolean* zde odpovídá *Bool*, *Integer* je pojmenován

stejně, *UnlimitedNatural* zase odpovídá typu *Real* a nakonec *String* poli znaků - *[Character]*. Stačí tedy během generování kódu přizpůsobit jejich názvy.

Jazyk *e-PFL* neumožňuje nějakým způsobem zohlednit viditelnost jednotlivých prvků systému. Proto nemá smysl se zde zabývat typy viditelností, definovatelných pomocí jazyka *UML*.

Viditelnost by v jazyce *e-PFL* šla alespoň nějakým minimálním způsobem reprezentovat, pokud by byla u tohoto jazyka rozšířena syntaxe, týkající se definice modulů způsobem, jakým to je například provedeno v jazyce *Haskell* [2]:

```
module DataModule (select, Insert, Delete) where
import ConnectionModule
...
select x y =
...
```

Výpis 9: Export funkcí v jazyce Haskell

Pokud by zde jeden modul reprezentoval jeden balík, byly by exportovány funkce (uvedené v ukázce v závorkách za názvem deklarovaného modulu) typu *public*. Zbývající prvky (typu *private*, *protected* a *package*), by byly pro ostatní moduly nedostupné.

Musely by zde být také brány v potaz vztahy mezi prvky, umístěnými v různých modulech. Například při použití viditelnosti typu *protected* musí být daný *UML* prvek, reprezentovaný v kódu funkcionálního jazyka viditelný pro všechny prvky z něj dědící. Ty se mohou nacházet v jiných modulech.

10.2.1.1.3 Klasifikátory Klasifikátory, reprezentující třídy objektů mající nějaké vlastnosti, nemají ve funkcionálním jazyce *e-PFL* žádnou přímou reprezentaci. Nelze zde tvořit žádné instance či jejich předpisy jako v klasických objektově orientovaných jazycích. Proto ve vygenerovaném kódu půjde spíše o samotnou reprezentaci funkcí v klasifikátorech obsažených. Pokud bude chtít uživatel nějakým způsobem používat v kódu objektový přístup, bude jej muset podstatně změnit a přidat minimálně nějakou reprezentaci stavů systému.

Pro zpřehlednění a zjednodušení generovaného kódu by v tomto směru možná přispělo jazyku *e-PFL* menší přiklonění k objektovému paradigmatu. Velice by se zde hodilo doimplementování využití mechanismu typových tříd, používanému například v jazyce *Haskell* [2]. Ulehčilo by to například znázornění vztahů mezi objekty jako je generalizce, či realizace rozhraní.

V generovaném kódu proto budou tyto jednotlivé klasifikátory reprezentovány pouze svými funkcemi a vlastnostmi. Doplňkové informace o klasifikátorech budou přidány pomocí komentářů.

Mezi klasifikátory, jejichž vlastnosti budou v generovaném kódu reprezentovány, patří *Signal*, *Primitive Type*, *Data Type*, *Class* a *Component*. Nebude zde provedena generace funkcí rozhraní. Rozhraní budou zohledněna až během jejich realizace, viz vztahy 10.2.1.1.7.

Každý klasifikátor bude ve výsledném kódu reprezentován komentářem, obsahujícím typ klasifikátoru, jeho název a cestu v rámci stromu *UML* modelu. Navíc zde budou vy-

psány názvy jednotlivých diagramů chování, asociovaných s tímto prvkem (viz načítání *UML* modelu 8.1). Ty může uživatel během doplňování vygenerovaného kódu případně využít jako průvodce.

Jedinou výjimkou ve způsobu reprezentace zde bude prvek *PrimitiveType*. Ten bude ve výsledném kódu reprezentován jako nový datový typ pomocí klíčového slova *data*, stejně jako výchozí primitivní typy *Integer*, *Bool* apod. Tato deklarace bude přidána pod odpovídající komentáře, viz fragment kódu níže:

```
    ...
----SIGNAL----
--name > CommunicationSignal
--path > System_Communication
    ...
----CLASS----
--name > CommunicationEndPoint
--path > System_Communication
    ...
----PRIMITIVE TYPE----
--name > SignalArray
--path > System_Communication
data System_Communication_SignalArray
    ...
```

Výpis 10: Reprezentace klasifikátorů v kódu jazyka e-PFL

10.2.1.1.4 Vlastnosti klasifikátorů Vlastnosti jednotlivých klasifikátorů jsou v jazyce *UML* reprezentovány prvky, dědícími ze tříd *StructuralFeature* (např. *Property* a tento prvek rozšiřující *Port*) a *BehavioralFeture* (*Reception* a *Operation*).

Kvůli nemožnosti reprezentovat vhodným způsobem v jazyce *e-PFL* třídy objektů bude do názvů vlastností klasifikátorů zahrnuta také jejich cesta ve stromu *UML* modelu.

Vlastnosti klasifikátorů, zmíněné výše budou v generovaném kódu reprezentovány následovně:

```
    ...
-- // Properties
-- name
-- type > [Character]
System_Users_name = "Jan Novák"
-- age
-- type > Integer
System_Users_age = 38
-- children
-- type [Character]
System_Users_children = -- length > *
    ...
```

Výpis 11: Reprezentace vlastností klasifikátorů v kódu jazyka e-PFL

Je zde vidět zakomponování cesty k dané vlastnosti (*System_Users_...*), která je potřebná pro odlišení operací a atributů daného klasifikátoru od případně stejně pojmenovaných

atributů a operací jiných klasifikátorů, obsažených ve stejném modulu. Uživatel díky tomu také může přímo vidět, o kterou vlastnost či operaci se jedná.

Atributy budou v generovaném kódu uvedeny pomocí komentáře - - // *Properties*. U každého z nich bude dále v komentáři o řádek níže uveden jeho čistý název, bez zakomponování cesty k němu v rámci stromu *UML* modelu. O řádek níže bude ještě informace o typu atributu.

V případě existence výchozí hodnoty atributu je tato hodnota také do výsledného kódu vložena.

Při generování atributu mimo jiné musí být zohledněno, zda se jedná o pole, nebo ne. Pokud se bude jednat o pole, bude k danému atributu přidán komentář o jeho případné délce (viz atribut *children* v ukázce výše).

Pokud se zde bude jednat o atribut, reprezentující referenci na nějakou instanci, bude tento atribut pouze reprezentován komentářem.

Prvky dědící z typu *BehavioralFeature* (*Operation* a *Reception*), obsažené v daném klasifikátoru zde budou reprezentovány jako obyčejné funkce. V syntaxi jazyka *e-PFL* je nelze nějak odlišit. Pro jejich odlišení bude pouze na začátku jejich definice komentářem uvedeno, zda se jedná o operaci či o přijetí signálu.

Samotná transformace operací je velice jednoduchá, nejprve bude uveden název operace a po něm názvy vstupních parametrů. Výstupní parametry budou bez znalosti vnitřní struktury odpovídají funkce uvedeny za komentář, na stejný řádek, jako definice funkce.

V jazyce *UML* lze definovat čtyři typy směru parametru - in, out, return a inout. Reprezentace prvních tří typů ve výsledném kódu je zřejmá, u vstupně výstupního typu bude v definici funkce daný parametr vygenerován jako vstupní i jako výstupní.

Výjimku při generování parametrů, zde podobně jako u atributů, tvoří parametry reprezentující typy, deklarované v rámci modelu. Jelikož samotný jazyk *e-PFL* jako funkcionální jazyk nedovoluje používat reference na objekty ve smyslu instancí objektově orientovaného programování, nemá smysl tento typ parametrů ve vstupních parametrech dané funkce žádným způsobem reprezentovat. V případě potřeby může být v těle funkce, vytvořeném uživatelem, zavolána odpovídající funkce, reprezentující potřebnou operaci dané třídy.

Nad samotnou definici funkce bude ještě uvedena informace o typech vstupních a výstupních parametrů a čistý název operace, bez zakomponované cesty pro odlišení od dalších funkcí.

```
    ...
-- // Operations
<<Integer::hours, Real::hourlyWage>> ClaculatePayment <<Integer::payment>>
System_Users_CalculatePayment hours hourlyWage = -- payment
    ...
-- // Receptions
    ...
```

Výpis 12: Reprezentace operací klasifikátorů v kódu jazyka e-PFL

10.2.1.1.5 Balíky Balík, reprezentující určitou podmnožinu prvků daného systému bude ve výsledném kódu nejjednodušší reprezentovat pomocí jednoho modulu. Ve funkcionálním programování modul reprezentuje určitou kolekci funkcí, typů a typových tříd. Umožňuje rozdělit komplexnější kód do menších částí pro lepší přehlednost, nebo případně dovoluje kus kódu, obsažený v daném modulu použít i v jiných projektech.

Problém zde ale nastává v případě zanoření balíků, v jednom definovaném modulu již další vytvořit nelze. Zanoření zde proto bude suplováno mechanismem importování modulů. Každý balík zde bude tvořit jeden modul. Pokud bude například v modelu zanořen balík A v balíku B, bude zde vytvořen modul A a modul B, kde modul A bude importovat modul B.

Kvůli lepší přehlednosti při případném větším počtu vygenerovaných modulů bude při tvorbě názvu generovaného modulu zohledněna i jeho cesta. Díky tomu dojde i k vyvarování se případů, kdy by byly do stejné složky projektu generovány soubory modulů se stejným názvem, ale kde každý z nich je umístěn v jiné části hierarchie *UML* modelu.

Reprezentace zanoření prvků v jazyce *e-PFL* nemá význam, vše je po importu globálně dostupné, Rozdělení do modulů má smysl pouze pro zpřehlednění a pro zviditelnění struktury původního *UML* modelu. Při provedení importu modulu do nějakého modulu projektu v jazyce *e-PFL* dojde k automatickému zpřístupnění funkcí obsažených v importovaném modulu všem ostatním modulům.

V tomto směru také nemá smysl zabývat se ostatními definovatelnými vztahy mezi balíky, jako *Package Import*, *Package Merge* a *Element Import*.

10.2.1.1.6 Znázornění zanořených prvků V jazyce *UML* lze do jednotlivých tříd, podobně jako do balíku, vkládat další, zanořené prvky (jako například další třídy, rozhraní apod.). Případné zanoření daného prvku lze v třídním diagramu znázornit buď přímým vložením potřebné třídy do jiné, či pomocí vazby *Containement Connection*.

Případy tříd a komponent, obsahujících ještě jiné prvky (mimo operace a vlastnosti) zde budou reprezentovány stejným způsobem jako balíky - pomocí samostatných modulů. každý z nich zde bude obsahovat kód, reprezentující jednotlivé vnořené prvky.

Každý takovýto modul bude uveden informací, že se jedná o reprezentaci prvku se zanořenými prvky. Po této informaci bude uvedena samotná definice třídy. Tato reprezentace bude provedena stejně jako u klasických klasifikátorů, viz 10.2.1.1.3. Po tomto kusu kódu budou následovat samotné reprezentace zanořených prvků.

10.2.1.1.7 Vztahy mezi prvky diagramu V textu dále budou rozebrány jednotlivé typy vztahů, které umožňuje třídní diagram znázornit a jejich využití při generování kódu.

Prvním z těchto vztahů je asociace. Ta znázorňuje využití funkčnosti jednoho klasifikátoru jiným klasifikátorem. Ve výsledku je tedy instance druhého klasifikátoru vlastností toho prvního a jedná se tedy o vlastnost, která je definována ve třídě prvního klasifikátoru. Jelikož zde v jazyce *e-PFL* možnost reprezentovat nějakým způsobem instance není, a jelikož jsou všechny funkce globálně dostupné, nemá smysl se touto vazbou zabývat.

Dalším vztahem, použitelným mezi prvky třídního diagramu je závislost - *Dependency*. Je zde reprezentována přerušovanou čarou a klasickou šipkou. Tento vztah je používán v případě dále nedefinované závislosti mezi dvěma klasifikátory. Kvůli nemožnosti přímo definovat v jazyce *e-PFL* třídy objektů, tak jak jsou známy z objektového paradigmatu, není zde třeba tento vztah během generování kódu brát v potaz.

To samé platí i pro vztahy, které vztah *Dependency* rozšiřují. K nim patří například *Usage* či *Abstraction*.

Zajímavé z hlediska generování kódu jsou dva vztahy rozšiřující třídu *Abstraction* - *Realization* a *Interface Realization*. Při provázání nějaké třídy s rozhraním či jinou třídou pomocí jednoho z těchto vztahů dojde během generování kódu k přidání funkcí daného rozhraní či třídy k funkcím tříd, na ně navázaných. Dojde také k přidání odpovídajících informačních komentářů (viz úryvek kódu níže).

Stejným způsobem bude proveden mechanismus dědění pomocí vztahu generalizace. Funkce, které nejsou při generalizaci upravovány a lze je tedy i volat z funkcí třídy, ze které daná třída dědí, bude moci uživatel případně po vygenerování smazat.

```
    ...
-- // ////////  Realization of > IEquatable
-- // Operations
<<Bool::isEqual>> Equals <<[Character]::a, [Character]::b>>
    ...
-- // ////////  Inherited Members
--From > General
    ...
```

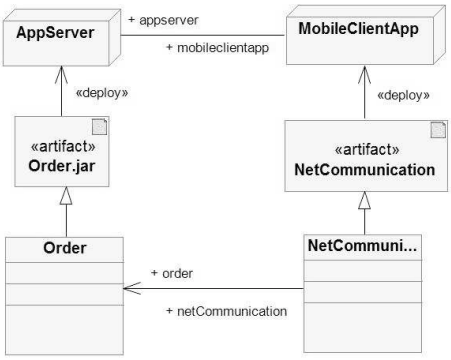
Výpis 13: Realizace rozhraní a generalizace v kódu jazyka e-PFL

Při výše zmíněné implementaci mechanismu typových tříd (viz 10.2.1.1.3) do jazyka *e-PFL*, by bylo využití těchto rozšiřujících vztahů mnohem názornější, přímější a pro uživatele o moc přehlednější.

10.2.1.2 Objekty UML modelu, umístěné ve více diagramech najednou V potaz musí být během generování kódu brány i objekty, které jsou modelovány ve více diagramech zároveň (viz 7 - přetahování objektů ze stromu *XMI* do určitého diagramu). Během generování kódu bude muset být také kontrolováno, zda náhodou již daný objekt nebyl někde vytvořen dříve. Pokud ano, bude generování kódu, reprezentujícího tento prvek přeskočeno.

10.2.1.3 Diagram objektů a diagram balíků Diagramy objektů a diagramy balíků lze v nástroji *Papyrus UML* vytvořit pomocí třídního diagramu, budou tedy i jako třídní diagramy brány během samotného generování kódu a transformace jejich jednotlivých prvků bude provedena způsobem popsaným výše.

10.2.1.4 Diagram profilu Prvky diagramu profilu, jako je například balík označený stereotypem «*Profile*», či vztah nazvaný *Extension* nemají z hlediska generování kódu urče-



Obrázek 7: Diagram nasazení

ného pro simulaci vestavného systému smysl, proto nebudou při samotné transformaci brány v potaz.

10.2.1.5 Komponentní diagram Tento diagram, znázorňující jednotlivé komponenty systému obsahuje až na pár výjimek stejné prvky, jako třídní diagram. Samotná komponenta zde reprezentuje jakýsi kontejner, obsahující další prvky, vykonávající určitou funkčnost a která poskytuje nějaká rozhraní a jiná zase pro svoji funkčnost vyžaduje.

Z pohledu tvorby prvků jazyka *e-PFL* má smysl ji brát pouze jako jakýsi kontejner. V rámci jedné komponenty se například může nacházet několik paralelně pracujících jednotek, které má smysl pomocí jazyka *e-PFL* simulovat a ve výsledném kódu reprezentovat samostatně. Proto bude uživatel během transformace vyzván, aby z těchto zanořených prvků ty aktivní případně vybral.

Při tvorbě pomocného kódu zde také nemá smysl jít v analýze jednotlivých komponent do hloubky. Jednotlivé obsažené prvky jiným způsobem než výčtem jejich vlastností a funkcí reprezentovat nelze. Kvůli zpřehlednění ale má smysl pro komponentu obsahující další části vytvořit samostatný modul (viz 10.2.1.1.6).

10.2.1.6 Diagram nasazení Diagram nasazení, jenž je určen pro zobrazení implementačního pohledu na systém, konkrétního nasazení jednotlivých prvků během vykonávání a může zobrazovat například závislost na jednotlivých platformách. Je používán spíše v pozdějších fázích vývoje systému. Z pohledu využití při generování jazyka *e-PFL* moc velký význam nemá. Například na obrázku 7 artefakt *Order.jar* reprezentuje třídu *Order* a je nasazen na uzlu *AppServer*. Pro generování kódu jazyka *e-PFL* je zde ale důležitá pouze třída *Order*, další prvky lze zanedbat. Co se týče komunikace mezi jednotlivými uzly, kterou obstarává vazba *CommunicationPath*, ta umožňuje přenos dat mezi jednotlivými artefakty nasazenými na různých uzlech. Pokud dané třídy, reprezentované artefakty potřebují mezi sebou komunikovat, na vyšší úrovni abstrakce - v třídním či komponentním diagramu - je tato komunikace reprezentována klasickou asociací (viz obrázek 7). Při tvorbě kódu lze tedy *CommunicationPath* zanedbat. Z tohoto typu diagramu budou brány v potaz pouze prvky, které má tento diagram společné s třídním a komponentním.

10.2.1.7 Kompozitní diagram Tento diagram, přidaný ve verzi 2 jazyka *UML*, umožňuje vytvořit detailnější pohled na vnitřní strukturu klasifikátorů a jejich vzájemnou spolupráci. Protože se jedná o nový diagram a proto není ještě tolik používán, je při procesu transformace kladen důraz na používanější diagramy. Při transformaci z něj budou použity pouze prvky, které má společné s ostatními strukturálními diagramy.

10.2.1.7.1 Diagram spolupráce Pomocí prvků, obsažených v kompozitním diagramu lze vytvořit diagram spolupráce. Tento diagram, umožňující pohled na vzájemnou spolupráci jednotlivých instancí, tvořících systém by mohl být užitečný při generování prvků jazyka *e-PFL*. Instance, spolupracující pro dosažení nějakého cíle systému zde mohou tvořit paralelně pracující jednotky. Zde bude důležité určit, které z nich to jsou a také blíže specifikovat jednotlivé komunikační kanály, propojující jednotlivé funkční jednotky. Jedna vazba typu *Connector*, propojující dvě části systému může hostit několik komunikačních kanálů. Co se týče aktivních částí systému, zde můžou pomoci stereotypy *UML* profilu *MARTE*. U komunikačních kanálů bude zapotřebí pomoc uživatele, jedná se totiž o dost specifickou vlastnost jazyka *e-PFL*.

10.2.1.7.2 Diagram vnitřní struktury Dalším diagramem, který lze vytvořit pomocí prvku kompozitního diagramu je diagram vnitřní struktury. Umožňuje vytvořit detailnější pohled na vnitřní strukturu klasifikátoru, než například diagram třídní. I zde nastává problém s určením aktivních části modelovaného klasifikátoru, které by mohly případně tvořit funkční jednotky jazyka *e-PFL*. V samotném klasifikátoru mohou být obsaženy instance jiných klasifikátorů, které zde také mohou tvořit funkční jednotky. I zde bude problém definovat jednotlivé komunikační kanály. V případě využití tohoto diagramu během transformace bude i zde zapotřebí spolupráce uživatele.

10.2.2 Pohled na chování

10.2.2.1 Diagram aktivit a stavový diagram Stavový diagram umožňuje zachytit stavy objektu a jednotlivé přechody mezi těmito stavy. Jelikož se v něm pracuje přímo s objekty, bude těžké definovat nějaký postup automatické transformace na kód ve funkcionálním jazyce a to z důvodu nemožnosti nějakým jednoduchým způsobem jednotlivé stavy v samotném kódu reprezentovat (na rozdíl od imperativních jazyků).

Je zde možnost pokusit se o definici jakýchsi stavů prvků jazyka *e-PFL*, jako je proces či zařízení. S tím by ale musel být uživatel podrobně seznámen a musel by je brát v potaz během tvorby diagramů. Ve většině případů ale půjde o využití jazyka *e-PFL* během fáze analýzy tvorby vestavného systému a dále už nemusí být použit, proto by bylo také zbytečné pro vytvoření modelu systému v tomto jazyce vytvářet speciální diagramy.

Postupná transformace stavového diagramu na určitou část výsledného kódu s pomocí uživatele by šla například zakomponovat do vytvářeného nástroje, nastává zde však otázka, zda samotné procházení jednotlivými okny spolu s ostatními kroky transformace nezabere nakonec vice času, než přímé napsání samotného programu.

K podobnému problému dochází u aktivitního diagramu, který je zvláštním případem stavového. Za samotnou akcí, zobrazenou v diagramu mohou stát další, vnořené akce. Nastává zde také problém stanovení, co je akce. Akci může tvořit například tvořit volání funkce, získání hodnoty nějakého atributu, určitý krok uvnitř funkce, či vyvolání nějaké události.

Zde také bez nějaké přesné definice, podle které by se uživatel musel během tvorby diagramu řídit, nebo nějakého delšího průvodce transformací automaticky mnoho získat nelze.

Výše uvedené diagramy jsou z pohledu automatické transformace *UML* diagramů na kód příliš abstraktní.

10.2.2.2 Sekvenční diagram Sekvenční diagram umožňuje znázornit důležité scénáře, týkající se interakce jednotlivých instancí částí systému během jejich životního cyklu. Samotné instance tříd systému jsou zde reprezentovány životními drahami objektu, mezi kterými probíhá komunikace pomocí zasílání zpráv.

Tento typ *UML* diagramu umožňuje znázorňovat určité scénáře chování systému a jejich alternativy. Při znázorňování složitějších algoritmů lze dojít do bodu, kdy bude vizuální znázornění pomocí tohoto diagramu složitější, než jeho reprezentace v kódu nějakého jazyka.

Lze jej ale zpřehlednit například pomocí interakčních rámců, ty budou rozebrány podrobněji níže.

10.2.2.2.1 Pohled ze strany jazyka e-PFL Tento diagram bude užitečný z pohledu generování těl funkcí a procesů. Z pohledu jazyka *e-PFL* zde bude důležité rozlišovat, zda samotné životní dráhy reprezentují procesy zařízení jazyka *e-PFL*, nebo se jedná o pomocné funkce.

Pokud bude například jedna životní dráha reprezentovat provádění určitého procesu, bude zapotřebí rozlišovat mezi voláním funkcí z odpovídající životní čáry a voláním procesu zařízení.

Problém zde nastává, pokud se bude jednat o volání procesu. Procesy jednotlivých zařízení jsou zde prováděny najednou, proces v jazyce *e-PFL* je jednolitý kus kódu, který čeká na doplnění všech vstupních parametrů a během svého provádění s ostatními procesy nekomunikuje, ale až na konci vrátí výstupní parametry. Až poté mohou ostatní procesy nějakým způsobem na vytvořený výsledek reagovat. V případě volání procesu jiným procesem zde proto bude muset být provedeno navázání doplňkových komunikačních kanálů, viz část zabývající se procesy 10.1.3.

Dalším případem, který je nutné zohlednit při generování kódu je volání operace, která byla vybrána jako vestavný proces, pomocnou funkcí. Tento případ bude muset případ od případu ošetřit sám uživatel, například vhodnou funkcí, fungující jako prostředník.

10.2.2.2.2 Interakční rámce Jednotlivé interakční rámce mohou velmi pomoci při nastínění tvorby kostry větvení a cyklů uvnitř jednotlivých procesů a funkcí. V textu

níže budou postupně po jednom rozebrány a bude u nich uveden případný přínos pro generování kódu.

ref - Tento rámec v podstatě označuje odkaz na jinou interakci. Tato interakce bude také zahrnuta do tvorby kostry větvení vnitřní struktury daného procesu či funkce.

loop - Tento rámec umožňuje označit sérii opakujících se zpráv a také případný počet opakování. V jazyce *e-PFL* lze definovanou smyčku zobrazit tvorbou pomocné funkce, obsahující rekurzivní volání.

alt, opt - Tyto druhy rámců umožňují v sekvenčním diagramu zobrazit konstrukci *if..then..else*. Tato konstrukce je v syntaxi jazyka *e-PFL* zahrnuta.

strict - Zprávy, obsažené v tomto rámci musí být zaslány v pořadí, ve kterém jsou zobrazeny.

neg - Omezuje zprávy, provádějící chybnou, či neproveditelnou sekvenci volání. Při generování kódu bude tato část ignorována.

par - Označuje paralelní provádění kódu. Tento rámec by mohl mít význam při tvorbě paralelních procesů, bude zde ale vždy záležet na aktuálním kontextu.

U zbývajících rámců nelze definovat nějaké obecné využití během generování kódu, jejich využití bude záležet hlavně na okolních podmínkách.

10.2.2.3 Diagram případů užití a zbývající diagramy chování

Diagram případů užití nemá z hlediska generování kódu žádný význam.

Jelikož není v nástroji *Papyrus UML* implementována podpora tvorby diagramu časování, diagramu přehledu interakcí a komunikačního diagramu, také nebyly do úvah o transformaci zahrnuty.

10.3 Využití profilu MARTE během transformace

Profil *MARTE* při modelování nabízí širokou škálu stereotypů umožňujících popsat různé aspekty vestavného systému (je jich v něm obsaženo více jak sto). Mohou být nápomocné při generování kódu v jazyce *e-PFL*. Je zde nutné vybrat množinu stereotypů, které bude mít během transformace smysl použít. Proto je nutné nejprve definovat pravidla, podle kterých budou tyto vhodné stereotypy vybírány.

Při aplikaci stereotypů tohoto profilu na prvky *UML* modelu vestavného systému nemusí být vždy použity ty nejvhodnější, proto je vždy nutné ještě případně počítat s pomocí uživatele.

Vhodné stereotypy by měly poskytovat dodatečné informace - ty které není schopen poskytnout čistý jazyk *UML*.

Prvním typem informací, usnadňujících transformaci je informace o tom, zda se jedná o samostatnou, paralelně pracující část systému, která obsahuje nějaké funkce. V tomto

směru se dá množina vhodných kandidátů omezit na stereotypy, které jsou aplikovatelné na klasifikátory, tj. prvky, obsahující nějaké operace a vlastnosti (viz 10.2.1.1.3).

Všechny operace, obsažené v klasifikátoru nemusí automaticky reprezentovat poskytované funkce funkční jednotky. Mohou také reprezentovat pouze pomocné operace, se kterými nemusí okolí přijít do styku.

Jednotlivé funkční jednotky musí mezi sebou nějakým způsobem komunikovat. Zde může nastat problém při zjišťování jednotlivých komunikačních kanálů. Jeden komunikační kanál reprezentovaný v *UML* diagramu zde může například reprezentovat několik komunikačních kanálů, charakteristických pro jazyk *e-PFL*. Jako příklad lze uvést například situaci, znázorněnou na obrázku výdejního automatu 6 v části 10.1.3. Dva komunikační kanály, nacházející se mezi funkčními jednotkami *Controller* a *Serving* (kanály *SoldItem* a *Choice*) mohou být reprezentovány v *UML* diagramu jako jediný komunikační kanál. Komunikační kanál zde také může reprezentovat například volání nějaké operace.

Může zde také nastat případ, ve kterém může být samotný komunikační kanál, znázorněný v *UML* diagramech, reprezentován jako samostatná funkční jednotka, poskytující nějaké funkce.

V následujícím textu budou postupně probrány jednotlivé části profilu *MARTE* a popsány jednotlivé použitelné stereotypy a jejich význam při transformaci. Byly zde vybírány hlavně ty stereotypy, které má smysl využít v první části transformace, při využití strukturálních prvků jazyka *UML*.

10.3.1 MARTE Foundations

Stereotypy, použitelné pro získání potřebných informací o vestavném systému se nacházejí až v balíku *Generic Resource Modeling*. V podstatě se jedná o všechny stereotypy, dědící ze stereotypu *Resource*. Ten reprezentuje jakýsi abstraktní prostředek a může případně reorezentovat funkční jednotku vestavného systému. Mezi vhodné stereotypy zde patří *StorageResource*, *ClockResource*, *ComputingResource* či *DeviceResource*. Jejich názvy mluví samy za sebe.

Stereotypy z balíků *Timing*, *NFPs* a *Alloc* nemá smysl uvažovat, popisují systém na nižší úrovni abstrakce a informace v nich obsažené při tvorbě modelu využít nelze.

10.3.2 MARTE Design Model

V této části se nacházejí vhodné stereotypy v balíku *High-Level Application Modeling*. Vhodnými kandidáty jsou stereotypy *PpUnit* a *RtUnit*, které umožňují definovat pasivní, respektive aktivní části modelovaného systému a lze je tedy využít při definování funkčních jednotek jazyka *e-PFL*.

Pro určení procesů těchto funkčních jednotek jsou vhodné stereotypy *RtAction* a *RtSerivce*, které lze aplikovat na prvky, dědící z typu *BehavioralFeature*. K nim patří třídy *Operation* a *Reception*.

Je zde také nutné brát v potaz to, že by prvek, rozšířený pomocí stereotypu *PpUnit* mohl být také reprezentován aktivní funkční jednotkou.

Další v této části umístěný balík, *Detailed Resource Modeling*, také obsahuje vhodné stereotypy. Tento balík je rozdělen na dvě části (*Software Resource Modeling* a *Hardware Resource Modeling*) ale na vysoké úrovni abstrakce modelu vestavného systému, vytvořeném pomocí jazyka *e-PFL* nás však nemusí zajímat, zda se jedná o prvky ze softwarové, či hardwarové části.

Pro tyto dva balíky platí, že použitelné stereotypy rozšiřují vhodné stereotypy, umístěné v balíku *Generic Resource Modeling*. Stačí tedy pouze zjistit, zda daný stereotyp nedědí z vhodného nadřazeného stereotypu.

Mezi přijatelné stereotypy ze softwarové části patří například *SwCommunicationResource*, *SwConcurrentResource* či *MessageComResource*. Z hardwarové části sem patří *HwComputingResource*, *HwCoolingSupply*, *HwDMA* či *HwProcessor*.

10.3.3 MARTE Analysis Model

V této části, skládající se ze tří balíků, se také nacházejí vhodné stereotypy, které dědí z těch, obsažených v balíku *Generic Resource Modeling*. Jsou pouze obohacené o informace, které jsou potřebné pro jednotlivé typy analýz. O využití těchto informací na abstraktní úrovni, na které je vytvořen spustitelný model systému pomocí jazyka *e-PFL*, nemá smysl uvažovat.

10.3.4 Závěr

Ve výsledku zde na této úrovni abstrakce, při generování struktury modelu vestavného systému, má smysl pouze využití stereotypů, které jsou obsaženy v balících *Generic Resource Modeling* a *High-Level Application Modeling*. Při výběru vhodných reprezentantů je také zapotřebí brát v potaz situaci, kdy může být na některý prvek systému aplikován stereotyp, popisující jej jako pasivní část, ale v generovaném modelu má smysl reprezentovat jej funkční jednotkou.

10.4 Výsledný postup transformace

Samostatný proces transformace musí být co nejvíce automatický. Na druhou stranu ale musí uživateli dovolit samotnou transformaci vhodně upravovat podle jeho představ. V tomto případě ale nesmí samotný postup transformace sklouznout do sekvence velkého množství kroků, kdy by mohl proces transformace trvat déle, než klasické manuální napsání kódu odpovídajícího systému.

Systém nebude pomocí jazyka *UML* s největší pravděpodobností vytvořen do nejmenších detailů, pro různé situace budou použity různé diagramy, popisující důležité vlastnosti systému a situací během jeho běhu. Během vytváření modelu systému může také dojít k nechtěným nejednoznačnostem, nebo situacím, které nepůjde důkladně pomocí jazyka *UML* popsat. Proto jsou během transformace brány v potaz hlavně strukturální diagramy, diagramy chování budou brány pouze jako doplněk (viz sekvenční diagram 10.2.2.2.2).

Uživatel bude muset nejprve vybrat z projektu diagramy, které popisují problém, který má být simulován pomocí jazyka *e-PFL*. Uživatel může například chtít simulovat pouze nějakou podmnožinu navrhovaného vestavného systému.

Následně bude provedeno automatické prohledání struktur obsažených v diagramech a vybrány všechny prvky, které mohou komponenty vestavného systému reprezentovat (prvky typu *Classifier*, viz 10.1.1).

10.4.1 Výběr diagramů, komponent a jejich následná úprava

Uživateli bude poté zobrazen seznam vhodných kandidátů komponent systému i s jejich vlastnostmi. Zde bude umožněno uživateli jednotlivé komponenty upravovat. To znamená vybrat vhodné reprezentanty procesů z dostupných vlastností komponenty, určit typ přístupu k výběru procesu určeného k provádění (viz 2.2), typ generovaného kódu (*Emulator*, *Hume*, *MicroNET* - viz 2.2), a případně ještě v případě potřeby upravit název dané komponenty či vlastnosti.

V tomto kroku by mělo být také umožněno vytvořit více reprezentantů určité komponenty modelovaného systému. Tento počet totiž nemusí být z dostupných diagramů jasně zřetelný.

Další krok bude zaměřen na samotné provázání jednotlivých procesů komponent pomocí komunikačních kanálů. Zde bude moci uživatel provázat vstupní a výstupní parametry jednoho procesu či je svázat se vstupními nebo výstupními parametry jiného procesu (viz 10.1.3).

10.4.2 Extrakce prvků Device a anotací Rename

V této fázi jsou již dostupné informace o jednotlivých komponentách, obsažených procesech a jejich vzájemném provázání. Z nich již lze vytvořit samotné prvky *Device*. Proces jejich získání bude následující:

Nejprve jsou jednotlivé komponenty rozděleny do dvou množin podle definovaného přístupu k procesům (viz 2.2).

Následně jsou v těchto dvou množinách komponenty roztříděny podle počtu obsažených procesů.

V každé takto získané množině je vybrána jedna komponenta, vytvořena nová množina, do které je umístěna, a názvy jejich jednotlivých procesů jsou porovnávány s názvy procesů zbývajících komponent. Pokud jsou názvy procesů nějaké komponenty identické s názvy procesů vybrané komponenty, je tato komponenta přidána do této nové množiny. Takto jsou porovnány všechny komponenty a na konci jsou vytvořeny množiny, které obsahují komponenty s identickými názvy procesů.

V každé takové množině ale mohou existovat procesy, které sice mají stejný název, ale mohou v porovnání s ostatními obsahovat rozdílně vstupní a výstupní parametry (nejprve budou porovnávány jejich počty, poté jednotlivé typy). Kvůli tomu jsou ještě všechny procesy se stejnými názvy v dané množině komponent zkontrolovány i v tomto směru. V případě neshody pro každý takový případ vytvořena nová množina, do které jsou pak ještě případně přidány další, ekvivalentní komponenty.

Ve výsledku jsou zde získány množiny komponent, kde každá reprezentuje jedno zařízení *Device*. Každé zařízení má odpovídající přístup k výběru procesu k provádění (*Fair* či *Unfair*) a seznam procesů s odpovídajícími vstupními a výstupními parametry.

Zbývá zde ještě vyextrahovat anotace *"rename"* (viz 2.2). To bude provedeno tak, že nástroj z každé množiny komponent daného zařízení vybere jednoho reprezentanta, který bude brán jako výchozí a pro každou komponentu, obsahující nějaký proces s rozdílným pojmenováním vstupního či výstupního parametru, bude pro danou komponentu vytvořena nová anotace reflektující tento rozdíl. Pro každé z těchto zařízení je také automaticky vygenerován název. Uživatel by měl mít následně možnost tyto názvy podle potřeby ještě upravit.

10.4.3 Generování kódu

V tomto bodě byly od uživatele získány všechny potřebné informace a nyní bude proveden samotný proces generování kódu jazyka *e-PFL*. Nejprve bude vygenerován kód modulu, který bude obsahovat definici jednotlivých zařízení, jejich procesů, komponent a metoda main. Budou zde umístěny také vlastnosti a funkce, které nebyly při tvorbě komponent označeny jako budoucí procesy.

Pro zpřehlednění budou jednotlivé definice zařízení a komponent uvedeny komentářem. Tím bude také zvýrazněno oddělení případných pomocných funkcí daného zařízení.

Je zde nutné také brát v potaz případy, kdy se v parametrech operace, označené jako budoucí proces nachází parametry reprezentující nějaké objekty tříd, obsažených v modelu. Kvůli nemožnosti vhodným způsobem reprezentovat samotné instance tříd budou v definici procesu uvedeny názvy těchto parametrů, nebude ale uveden jejich typ. Uživatel jej bude muset vhodným způsobem nahradit, případně je smazat. Pokud se bude jednat o objekty tříd, vybraných za reprezentanty funkčních jednotek systému, budou tyto parametry odstraněny a funkcionalitu, požadovanou od daného objektu bude muset uživatel reprezentovat vhodným provázáním s odpovídajícími procesy (viz 10.1.3 Procesy).

Musí být také ošetřeno případné vkládání procesů či pomocných funkcí a vlastností se stejným jménem. Pokud bude v daném kontextu již obsažen proces se stejným názvem, bude k na začátek názvu nově přidávaného doplněn název zařízení, ke kterému patří. Stejný způsob bude použit i u pomocných funkcí a vlastností, zde ale bude provedeno přidání názvu komponenty.

Následně bude pro každý vybraný *UML* diagram vytvořen jeden modul s prvky obsaženými v daném diagramu. Pro všechny prvky diagramů, zachycující stromovou strukturu systému (jako například balíky, třídy či komponenty - viz 10.2.1.1.6) jsou také vytvořeny samostatné moduly.

V této části generování kódu mohou být také využity diagramy chování (hlavně sekvenční diagramy) pro vytvoření kostry vnitřní struktury jednotlivých procesů a pomocných funkcí.

11 Implementace transformace

V této části bude nejprve popsána struktura kódu celého nástroje a poté provádění transformace. Samotná implementace transformace bude popsána nejprve z pohledu uživatele a vytvořeného uživatelského rozhraní, až pak z hlubšího pohledu na úrovni kódu.

11.1 Struktura nástroje

Nástroj byl implementován v jazyce C# ve vývojovém prostředí *Microsoft Visual Studio 2010*. Uživatelské rozhraní bylo vytvořeno za pomocí jmenného prostředí *Windows.Forms*, které patří pod technologii .NET.

Tento nástroj se skládá z knihovny *Loader.dll*, mající na starost načtení modelu vestavného systému popsaného v jazyce *UML*, dále knihovny *MARTE.dll*, reprezentující strukturu stereotypů a pomocných prvků *UML* profilu *MARTE* a knihovny *Transformation.dll*, mající za úkol samotný proces transformace.

Interakci s uživatelem obstarává uživatelské rozhraní implementované ve spustitelném souboru *UML2EPFL.exe*.

11.2 Uživatelské rozhraní

11.2.1 Hlavní okno

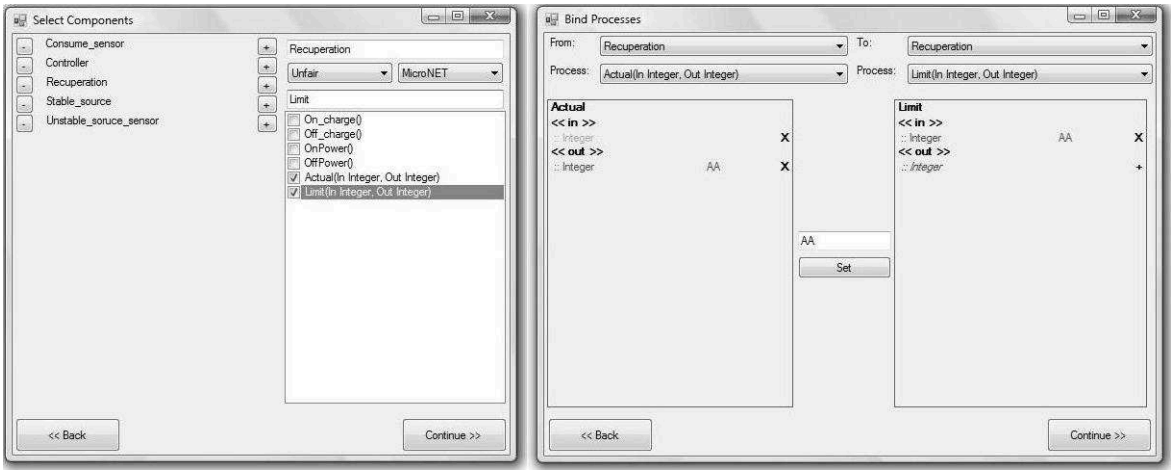
Hlavní okno tohoto nástroje je tvořeno hlavní nabídkou a ovládacím prvkem *TabControl*, který je určen pro zobrazení kódu jednotlivých modulů vestavného systému. Samotný kód je zobrazován pomocí ovládacího prvku *EditTab*, rozšiřujícího komponentu *TabPage*. Tato rozšířená komponenta obsahuje komponentu *RichTextBox*, která umožňuje editaci vygenerovaného kódu. Pro lepší viditelnost vygenerovaného kódu byl font prvku RichTextBox změněn na *Courier New* a velikost písma na 10 bodů. Ten je také například používán pro zobrazení kódu v nástroji *PSPad* či sadě *Visual Studio*.

Hlavní nabídka je tvořena pomocí čtyř tlačítek, nebylo zapotřebí vytvářet kvůli čtyřem možnostem rolovací menu. Tato nabídka obsahuje tvorbu nové transformace z jazyka *UML* do jazyka *e-PFL*, dále otevření již dříve vygenerovaného kódu, uložení aktuálně vygenerovaného kódu a nakonec nastavení adresáře, obsahujícího dříve vygenerované a uložené projekty a nastavení cesty k pracovní složce nástroje *Papyrus UML*.

Při volbě uložení aktuálního kódu se zobrazí dialogové okno *Save*, umožňující zadat název ukládaného projektu. Pokud není v adresáři projektů žádná složka se stejným názvem přítomna, je vytvořena nová složka se zadaným názvem a do ní uloženy všechny vygenerované moduly. Pomocí tlačítka *New* může uživatel provést transformaci *UML* modelu vestavného systému na kód jazyka *e-PFL*.

11.2.2 Okna provázející procesem transformace

Prvním oknem je uživatel nejprve vyzván k výběru souborů s příponou *uml* a *di2*, které daný vestavný systém reprezentují. Pokud uživatel zadá pouze soubor s příponou uml, je v daném umístění automaticky vyhledán také soubor **.di2* se stejným názvem. Po



Obrázek 8: Vlevo se nachází okno pro úpravu budoucích komponent a jejich procesů, vpravo je zobrazeno okno určené pro provázání jednotlivých procesů komunikačními kanály.

kliknutí na tlačítko *Continue* je pomocí knihovny *Loader* obsah souborů načten do paměti. Poté je zobrazeno okno umožňující výběr diagramů, které budou při generování kódu *e-PFL* použity. Výběr diagramů struktury a diagramů chování je oddělen. Po potvrzení vybraných diagramů jsou z diagramů struktury načteny všechny prvky *UML*, které mohou reprezentovat prvky jazka *e-PFL* jako je *Device* a *ESComponent* (viz 10.2.1.1.3). Z těch může uživatel vybrat konkrétní reprezentanty generovaného kódu. Tyto prvky jsou zobrazeny v ovládacím prvku *CheckedListBox*. U každého je pro zpřehlednění uvedena cesta v rámci jednotlivých diagramů. Uživatel může vybrat všechny prvky najednou zaškrtnutím volby *Select All*.

Po kliknutí na tlačítko *Continue* je zobrazeno okno umožňující definování vlastností jednotlivých komponent, pojmenované *Select Components* (viz obrázek 8). V levé části je seznam prvků vybraných v předešlém kroku. Každá komponenta je zde reprezentována ovládacím prvkem *Comp*, který tvoří dva tlačítka - jedno pro smazání komponenty ze seznamu, druhé pro vytvoření kopie dané komponenty a přidání této kopie do seznamu. Její jméno je automaticky vygenerováno tak, aby bylo unikátní, ale vychází z názvu původní komponenty. Název komponenty je uveden v prvku typu *Label*. Po kliknutí na vybraný prvek *Comp* jsou v pravé části zobrazeny jednotlivé vlastnosti vybrané komponenty.

V této části se nachází uživatelské rozhraní pro samotnou úpravu vygenerovaných komponent. Je zde ovládací prvek *TextBox* umožňující úpravu názvu komponenty, dva ovládací prvky *ComboBox* - jeden pro výběr typu přístupu výběru procesu, který má být aktuálně proveden (*Fair* a *Unfair*) a druhý pro výběr konstruktoru, použitého při definici samotné komponenty. Při úpravě názvu je automaticky upravován i název v ovládacím prvku *Comp*, který vybranou komponentu v seznamu reprezentuje. Pokud již

existuje komponenta s daným názvem, uživatel je na to upozorněn a provedená úprava názvu je vrácena zpět. Pod těmito ovládacími prvky se ještě nachází *CheckedListBox*, umožňující výběr jednotlivých procesů a vlastností vybrané komponenty. Uživatel zde ještě může upravit názvy budoucích procesů a pomocných funkcí a to výběrem daného procesu/vlastnosti v prvku *CheckedBoxList* a následnými potřebnými změnami v prvku *TextBox*, umístěného nad seznamem funkcí a vlastností.

11.2.3 Provázání parametrů procesů a tvorba komunikačních kanálů

Po kliknutí na tlačítko *Continue* v okně úpravy komponent dojde k zobrazení okna *Bind Processes* (viz obrázek 8), sloužícího k provázání jednotlivých procesů komponent. V horní části tohoto okna se nachází čtyři ovládací prvky typu *ComboBox* umožňující vybrat dva procesy ze dvou komponent a následně je provázat komunikačními kanály pomocí speciálního ovládacího prvku *OperationBinder*.

Tento ovládací prvek se skládá ze tří částí. Pravá a levá strana obsahuje samotné procesy účastnící se operace provázání, prostřední slouží k nastavení názvu aktuálně vytvořeného, nebo vybraného kanálu. Samotné procesy jsou zde reprezentovány názvem a seznamem vstupních a výstupních parametrů. Ty samotné reprezentují speciální ovládací prvky *ProcessParameter*.

Uživatel zde nejprve v horní části okna vybere první a druhou komponentu a následně jejich procesy, určené k provázání. Pak již pouze v prvku OperationBinder prováže jednotlivé parametry. Provázání je provedeno kliknutím na název nějakého parametru, umístěného v prvním či druhém procesu a následným kliknutím na druhý parametr, se kterým jej chce uživatel provázat. Tato komponenta umožňuje provázat parametr s parametry ostatních skupin. Jednu skupinu parametrů zde reprezentuje seznam vstupních nebo výstupních parametrů první či druhé komponenty. Celkově jsou zde čtyři. Lze tedy například provázat vstupní parametr prvního procesu s výstupem toho samého procesu nebo se vstupním, či výstupním parametrem druhého procesu. Provázat lze pouze parametry stejného typu.

Tato komponenta také umožňuje přidání nového parametru do vybraného procesu. Lze přidat pouze parametr jiného procesu. Pro jeho přidání musí uživatel kliknout na text «*IN*», respektive «*OUT*» procesu, kde má být nový parametr přidán a z druhého procesu vybere parametr jiné skupiny, který má být do dané skupiny prvního procesu přidán. Nově vytvořený parametr obsahuje pouze typ vybraného parametru, jeho název není nastaven, z pohledu dalšího zpracování nás již nemusí zajímat. Pokud je již existující parametr provázán s jiným, je název komunikačního kanálu převeden i do nově vytvořeného parametru. Ve výchozím stavu má text všech parametrů oranžovou barvu. Pokud dojde k provázání, je do tohoto textu přidán automaticky název provázání - komunikačního kanálu a barva textu je u obou koncových parametrů změněna na zelenou. Při provázání mohou nastat tři možnosti:

Žádný parametr není ještě s jiným provázán - je automaticky vygenerován nový název komunikačního kanálu a přidán k oběma parametrům.

Je nastaven komunikační kanál jednoho z vybraných parametrů - dojde k automatickému přidání názvu komunikačního kanálu do parametru, který jej ještě neobsahuje.

Jsou nastaveny komunikační kanály obou parametrů - Uživatel je vyzván k výběru jednoho z obsažených komunikačních kanálů. Vybraný kanál je nastaven i v druhém parametru.

Provázání lze zrušit kliknutím na symbol *X* u vybraného parametru. Při této akci je barva textu parametru změněna na oranžovou a je odstraněn název komunikačního kanálu. Při opětovném kliknutí na tento symbol je text parametru změněn na kurzívu a barva nastavena na červenou - daný parametr nebude při generování jazyka *e-PFL* použit. Opětovně jej použít lze kliknutím na symbol +. Tímto způsobem lze také ze seznamu parametrů úplně odstranit nově přidané parametry.

11.2.4 Potvrzení vygenerovaných prvků Device a zobrazení vytvořeného kódu

Po vytvoření potřebných vazeb mezi komponentami a kliknutím na tlačítko *Continue* dojde k automatickému vygenerování prvků *Device*, které jsou následně zobrazeny uživateli. K tomu slouží nové okno, obsahující seznam speciálních uživatelských prvků *DeviceSummary*. V každém je zobrazen automaticky vygenerovaný název nového *Device*, který lze upravovat, a způsob výběru procesu - *Fair* či *Unfair*. Dále se pod názvem nachází seznam komponent daného zařízení, ze kterých byl tento prvek *Device* vygenerován a vlevo jsou vypsány jednotlivé obsažené procesy. Po potřebných úpravách názvů může uživatel pokračovat kliknutím na tlačítko *Confirm*. Zde dojde k extrakci anotací *Rename*, vygenerování samotného kódu a jeho zobrazení v panelech hlavního okna.

11.3 Transformace na nižší úrovni

11.3.1 Sběr potřebných informací

Sběr potřebných informací zde probíhá na uživatelské úrovni, v uživatelském rozhraní aplikace (spustitelný soubor *UML2EPFL.exe*). Získané informace jsou obsaženy v objektu třídy *Context*.

Při prvním kroku transfrmace - výběru souboru *UML* modelu systému - je ihned po kliknutí na tlačítko *"Continue >>"* načten model systému do paměti a jeho reference předána do objektu, reprezentujícím kontext.

V dalším kroku, po výběru použitých diagramů, dojde ke vložení referencí vybraných diagramů do dvou kolekcí, umístěných v objektu kontextu - *Structure* a *Behavior*.

Po přechodu na další krok dojde k výběru vhodných kandidátů na funkční jednotky. Ti jsou reprezentováni objekty, jejichž třídy dědí ze třídy *Classifier* (viz 10.1.1). Tyto objekty jsou postupně vybrány průchodem stromovými strukturami strukturálních diagramů, obsažených ve výše zmíněné kolekci *Structure*. Pro korektní načtení všech vhodných

kandidátů zde slouží metody *LoadDeploymentDiagram*, *LoadDeploymentContent*, *LoadClassContent*, *LoadPackageContent*, *LoadComponentContent*, *LoadClassDiagram* a *LoadComponentDiagram*. Tito kandidáti jsou umístěni do kolekce tvořené objekty typu *UMLObject*. Objekty této třídy reprezentují prostředníky mezi objektem *UML* modelu a objektem, reprezentující danou komponenentu v jazyce *e-PFL*. Obsahují referenci na daného kandidáta, kolekci operací a vlastností, které mohou být použity jako procesy daného zařízení a také informace, které jsou zobrazeny uživateli v dalším kroku.

Vhodné procesy jsou zde reprezentovány třídou *Process*, obsahující vstupní a výstupní parametry společně s dalšími dodatečnými informacemi. Do vhodných kandidátů procesů jsou zahrnuty i operace a vlastnosti z nadřazených tříd a z realizovaných rozhraní.

Uživatel vybere v prvku uživatelského rozhraní typu *CheckedListBox* potřebné kandidáty na funkční jednotky. Reference na vybrané kandidáty jsou pak při přechodu na další krok transformace (úprava vybraných kandidátů) umístěny do objektu kontextu.

V tomto kroku je pro každého vybraného kandidáta vytvořen objekt typu *Comp*, který se stará o zobrazení daného kandidáta uživateli a obsahuje metody, umožňující s tímto kandidátem manipulovat. K nim patří například *AddActual*, starající se o zobrazení podrobností o vybrané komponentě, *AddComp* klonující aktuálně vybranou kompoentu či *RemoveComp*, odstraňující aktuálně vybranou komponentu ze seznamu kandidátů.

Tato třída také obsahuje reprezentanta prvku *ESComponent*, který bude následně použit během generování kódu a jehož vlastnosti budou v tomto kroku upravovány. V této části transformace jsou také ošetřovány případné úpravy názvu a je kontrolováno, zda již nebyla komponenta se stejným názvem vytvořena.

Po přechodu na další krok je uživateli nabídnuta možnost procesy vybraných komponent provázat pomocí komunikačních kanálů. Dva procesy, jejichž komunikační kanály chce provázat může vybrat pomocí čtyř prvků uživatelského rozhraní typu *ComboBox*. K jejich provázání zde slouží speciální prvek uživatelského rozhraní typu *OperationBinder*, viz 11.2.3. Tento prvek se také stará o automatické generování nové vytvořeného kanálu a o kontrolu, zda daný název nebyl v daném kontextu již použit.

Po provedení potřebných úprav a kliknutí na tlačítko *"Continue >>"* dojde k extrakci zařízení z obsažených komponent, K tomu slouží třída *ExtractDevices*. Samotný proces odpovídá postupu popsaném v části 10.4.2. Seznam takto vytvořených zařízení je poté předán objektu kontextu (kolekce *devices*). Jedno zařízení, reprezentované instancí třídy *Device* zde obsahuje seznam jednotlivých obsažených komponent a také informaci o typu přístupu k výběru procesu (*Fair* či *Unfair*).

Poté je uživateli zobrazen seznam vyextrahovaných komponent, u kterých ještě může jejich automaticky vygenerované názvy upravit I zde je kontrolováno, zda není vložený název již v kontextu obsažen. Po potvrzení všech potřebných úprav kliknutím na tlačítko *"Confirm >>"* dojde ke generování výsledného kódu.

11.3.2 Generování kódu

K provedení generování kódu zde slouží knihovna *TransformationEPFL*. Vstupním bodem ke generování je zde třída *Transformation*, které jsou v konstrukturu předány jako parametry všechny vybrané diagramy struktury a chování.

V této třídě je také vložena pomocná vnořená třída, nazvaná *Modules*. Ta obsahuje všechny potřebné informace o celkovém kontextu transformace. Obsahuje reference na hlavní modul, kolekci modulů, reprezentující jednotlivé diagramy a kolekci obsahující moduly, reprezentující další prvky *UML* modelu, jako například *Package* či *Class*, obsahující vnořené prvky. Obsahuje také dvě pomocné metody, umožňující vygenerovat jedinečný název modulu a jedinečný název generovaného prvku, jenž ještě nebyl v daném kontextu obsažen.

Generování kódu je zde vyvoláno zavoláním metody *Transform*. V té dojde k vytvoření modulu, reprezentujícího hlavní modul výsledného kódu a který bude obsahovat definice jednotlivých zařízení, jejich procesů, komponent a také funkci *main*. Poté jsou vytvořeny moduly, reprezentující jednotlivé diagramy, účastnící se transformace.

11.3.3 Tvoba struktury výsledného kódu

Struktura výsledného kódu je zde reprezentována pomocí objektů třídy *CodeStructure*. Ten obsahuje název daného segmentu kódu (reprezentujícího například funkci), řetězec *code*, obsahující kód reprezentující tento segment a řetězec *comment*, obsahující případné komentáře k tomuto segmentu. Dále jsou zde ještě obsaženy dvě kolekce objektů třídy *CodeStructure*, reprezentující segmenty kódu umístěné nad, respektive pod daným segmentem výsledného kódu.

Strukturu kódu jednoho modulu zde reprezentuje strom objektů typu *CodeStructure*. Ten je umístěn ve třídě *Code*, kde vlastnost *structure* reprezentuje jeho kořenový uzel. Je v ní dále obsažen název samotného modulu (*moduleName*), řetězec obsahující výsledný kód modulu (*code*) a referenci na uzel struktury, obsahující jednotlivé importy potřebných modulů (*imports*). Jeden objekt třídy *Code* zde reprezentuje strukturu kódu jednoho modulu, načítaného například pomocí výše zmíněné metody *Transform*.

Po vytvoření objektu tohoto modulu je zavolána metoda *GenerateStructure*, která má jako vstupní parametr objekt diagramu. V ní dojde k průchodu stromu daného diagramu, vložení jednotlivých prvků do struktury modulu a v případě potřeby vytvoření nového modulu - pokud bude například nalezen balík, či třída nebo komponenta, obsahující zanořené prvky. Pro každý typ prvku, obsahující zanořené elementy je zde obsažena metoda, načítající jeho obsah (metody *GeneratePackage*, *GenerateClass* a *GenerateComponent*).

V každém modulu, reprezentujícím prvek se zanořenými elementy je vložen import na nadřazený modul. Název každého nově vytvořeného modulu reprezentuje jeho cestu ve stromu *UML* modelu systému a zároveň tak vytváří jeho jedinečný název.

Pro vložení jednotlivých reprezentací prvků *UML* modelu do struktury modulu zde slouží metody *GetClass*, *GetComponent*, *GetSignal*, *GetPrimitiveType*, a *GetDataType*.

Jsou zde také umístěny pomocné metody *LoadBehavioralDiagrams*, *LoadAbstractions*, *InsertImport* a *CodeStructure*. Metoda *CodeStructureAdd* se zde stará o korektní vložení reference daného fragmentu reprezentovaného objektem typu *CodeStructure* do kolekce *CodeStructure_ID*, obsažené ve třídě *Modules*. Tato kolekce je následně využita během provádění asociace provázaných prvků (viz kapitola 11.3.6).

11.3.4 Hlavní modul

Hlavní modul, obsahující jednotlivé funkční jednotky, procesy, komponenty a funkci *main* je zde reprezentován třídou *MainCode*, dědící ze třídy *Code*. Oproti nadřazené třídě *Code* je zde navíc deklarován objekt typu *CodeStructure*, reprezentující zařízení, ve kterém jsou obsaženy fragmenty kódu, reprezentující jednotlivá zařízení. Je zde i další objekt typu *CodeStructure* reprezentující fragment kódu, obsahující funkci *main*. Výsledná struktura tohoto modelu je vytvořena již v konstruktoru této třídy.

11.3.5 Třída EpflCode

O vytvoření reprezentace *UML* prvků ve výsledném kódu se starají metody v této třídě obsažené. Jsou pomocí ní vytvořeny jak fragmenty pomocného kódu, tak i fragmenty hlavního kódu, obsaženého v modulu *Main*.

11.3.6 Asociace

Po vygenerování struktury jednotlivých modulů je ještě zapotřebí postarat se o prvky, které jsou s jednotlivými fragmenty provázány, ale informace o nich není přímo prezentována v dané části *UML* modelu. Jmenovitě se zde jedná o komentáře, omezení a vztahy abstrakce a realizace. Nelze je na odpovídající prvky navázat ihned, protože se během načítání ještě nemusí ve stromu struktury daného modelu nacházet. Proto jsou tyto prvky během generování struktury nejprve odkládány do struktury pojmenované *AssociatedObjects*.

Pro jejich provázání proto musí být ještě před samotným generováním kódu zavolána metoda *AssociateObjects*. Při provázání je zde velmi nápomocná kolekce *CodeStructure_ID*, o které byla již zmínka dříve.

11.3.7 Generování kódu

Nakonec je zapotřebí v metodě *Transform* pro každý modul zavolat metodu *GenerateCode*. O samotný průchod stromem kódu se zde stará metoda *GoThrough*. Strom zde procházen metodou *INORDER* a jednotlivé fragmenty jsou postupně vkládány do výsledného kódu. Výsledný kód modulu je pak obsažen ve vlastnosti *code*.

V metodě *Transform* je ještě nakonec provedeno vložení importů modulů reprezentujících diagramy do hlavního modulu. Nakonec je výsledný obsah jednotlivých modulů zobrazen uživateli.

12 Ukázkové příklady

Následující text obsahuje dva ukázkové příklady. Jeden je inspirován existujícím kódem jazyka *e-PFL*, druhý příklad reprezentuje několik diagramů *UML* modelu systému, skládajícího se z robotického ramene, komunikační části a řídící části.

První z nich poslouží k porovnání nástrojem vygenerovaného kódu s původním, na druhém příkladu bude ukázáno případné využití stereotypů profilu *MARTE*.

12.1 Příklad 1: Výdejní automat

Pro tvorbu tohoto příkladu posloužil kód jazyka *e-PFL*, nacházející se v souboru *vending.pfl* a reprezentující spustitelný model výdejního automatu. Jeho grafickou reprezentaci lze například vidět na obrázku 6. Tento příklad bude sloužit k porovnání původního kódu s vygenerovaným. V nástroji *Papyrus UML* jsem vytvořil model, reprezentující tento systém pomocí jazyka *UML*. V daném modelu každou funkční jednotku reprezentuje jedna třída. To neplatí pouze pro jednotky *testInput* a *testInput2*. Ty jsem zde reprezentoval pomocí jediné třídy. Obsahují totiž operace se stejnými vstupy a výstupy (jeden vstupní parametr typu *Integer* a dva výstupní, stejného typu) a také pole s hodnotami typu *Integer*, které obsahují vstupní hodnoty vestavného systému. Samotné vztahy mezi jednotlivými jednotkami jsou zde reprezentovány pouze pomocí vztahu *Usage*, nezabýval jsem se jimi podrobněji, samotné komunikační kanály totiž mohou být reprezentovány mnoho různými způsoby (viz procesy 10.1.3).

Pro předvedení extrahování anotace *rename* jsem během transformace upravil výsledný systém tak, aby byly sloučeny zařízení *testInput* a *testInput2*.

Při samotné transformaci zde bude v prvním kroku, po vybrání souboru projektu, nejprve vybrány diagramy, které budou použity během transformace. Zde se jedná pouze o diagram nazvaný *DefaultDiagram*. Po přechodu do dalšího kroku transformace je zapotřebí vybrat vhodné reprezentanty funkčních jednotek a vybrat jejich procesy. V tomto případě zde budou vybrány všechny zobrazené komponenty. Komponentu *testInput* zde bude zapotřebí naklonovat pomocí tlačítka "+" tak, aby mohly být ve výsledném kódu prezentovány dvě samostatné jednotky *testInput*.

V tomto kroku nebude kvůli předvedení extrahování anotace *rename* přejmenována operace *Operation_0*, takže ve výsledku budou tyto komponenty kvůli obsahu stejných prvků sloučeny v jedno zařízení.

U funkčních jednotek budou (až na vlastnosti *testInputData* u jednotek *testInput* a *testInput2* a *memoryData* jednotky memory) označeny všechny vlastnosti jako procesy jednotlivých funkčních jednotek.

Dále je zapotřebí nastavit přístup k výběru procesu k provedení u jednotek *memory* a *counter*. Ten u nich bude nastaven na *Unfair*.

V následujícím kroku je nutné provázat jednotlivé vstupní a výstupní parametry samotných procesů. Pro zjednodušení jsem do operací, umístěných v *UML* modelu umístil parametry, reprezentující všechny potřebné komunikační kanály. Samotné parametry operace ale nemusí reprezentovat všechny potřebné informace, viz Procesy 10.1.3.

Jednotlivé vygenerované názvy komunikačních kanálů pak bylo nutné přejmenovat tak, aby odpovídaly názvům v porovnávaném kódu.

V následujícím kroku je ještě nutné případně upravit názvy generovaných zařízení. Po jejich úpravě a potvrzení je nakonec zobrazen výsledný kód.

Fragmenty kódu, umožňující srovnání původního a vygenerovaného kódu jsou pro porovnání uvedeny níže. Celý vygenerovaný kód je umístěn v příloze ve složce *Tool_Workspace*.

```
import "Prelude.pfl"
    ...
testInputData = [0,2,2,2,1,1,1,1]

testInput :: a Integer -> (a Integer, input Integer)
testInput x = ( x+1, (x \% (length testInputData)) !! testInputData )

testInputDevice :: Device
testInputDevice = Process testInput

testInputData2 = [5,5,10,10,20,2,2,5]

testInput2 :: b Integer -> (b Integer, coin Integer)
testInput2 x = (x+1, (x \% (length testInputData2)) !! testInputData2 )

testInputDevice2 :: Device
testInputDevice2 = Process testInput2
    ...
main = (setData memoryData)`bl`(setActive (Value Active))`bl `(setA 0)`bl `(setB 0)`bl `(startDevice
    testInputDevice Simulator []) `bl `(startDevice testInputDevice2 Simulator []) `bl `(startDevice
    printer Simulator []) `bl `(startDevice controller Simulator []) `bl `(startDevice serving
    Simulator []) `bl `(startDevice counter Simulator []) `bl `(startDevice memory Simulator [])
```

Výpis 14: Původní kód výdejního automatu

```
import "Prelude.pfl"
import "DefaultDiagram.pfl"

-------- DEVICES --------

-------- DEVICE --------
Operation_0 :: A Integer -> (A Integer, input Integer)
Operation_0 A = (A, input)
deviceInput :: Device
deviceInput = Process Operation_0


-------- COMPONENTS of device_0 --------
testInput = ESComponent "deviceInput" Emulator

-------- auxiliary functions and properties
TestInputData = ()


testInput_0 = ESComponent "deviceInput" Emulator
```

```
-------- auxiliary functions and properties
TestInputData_0 = ()
    ...
-------- MAIN --------
main = (startDevice device_0 testInput  []) `bl` (startDevice device_0 testInput_0 [rename "A" "B",
       rename "input" "Coin", ]) `bl` (startDevice servingDevice Serving []) `bl` (startDevice
       controllerDevice Controller []) `bl` (startDevice CounterDevice Counter []) `bl` (startDevice
       MemoryDevice Memory [])
```

Výpis 15: Vygenerovaný kód výdejního automatu

Při porovnání s původním kódem je zde vidět navíc import modulu *"DefaultDiagram.pfl"*, který reprezentuje diagram, použitý během generování kódu a obsahuje pomocné definice. Nástroj zde také sloučil dříve zmíněná zařízení *testInput* a *testInput2* a vytvořil z nich jediné zařízení. Pro každou komponentu jsou zde pod komentářem obsahujícím text *"auxiliary functions and properties"* umístěny definice pomocných funkcí a vlastností. Jedná se o ty vlastnosti a funkce, které nebyly uživatelem označeny jako procesy.

Ve funkci *main* je při spouštění komponenty *testInput_0* pomocí funkce *startDevoce* vidět použití anotací *rename*. Odpovídající přejmenované komunikační kanály a jejich využití je názorně vidět na obrázku 6.

12.2 Příklad 2: Robot

Druhý ukázkový příklad reprezentuje systém, skládající se z řídící stanice, ovladače a robotického ramene. Tento příklad byl vybrán z dokumentu *"UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems"* [4], protože obsahuje názorné využití stereotypů profilu *MARTE*. Pokusím se zde předvést jejich využití během tvorby spustitelného modelu.

Model tohoto systému se skládá z několika diagramů. Ty jsou umístěny v příloze.

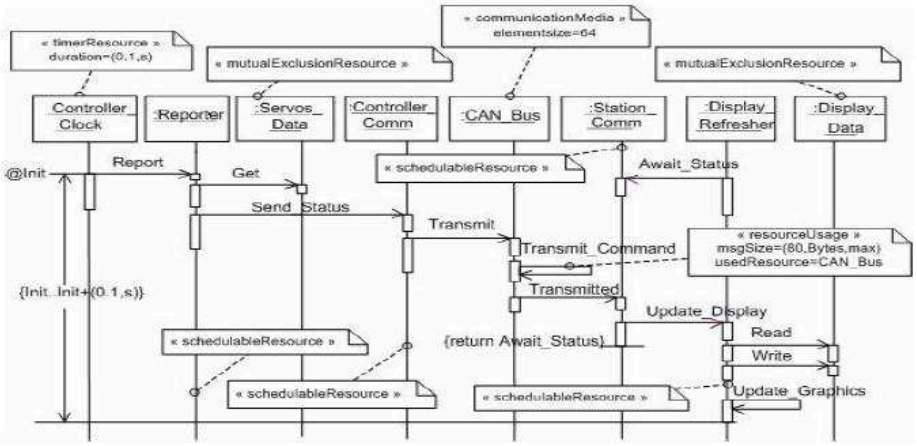
12.2.1 Scénář chování

Nejprve se pokusím popsat jeden scénář chování systému, reprezentovaný sekvenčním diagramem na obrázku 9.

Ten popisuje odeslání informací o stavu serv na řídící jednotku, kde jsou tyto data zobrazena.

Sekvence zpráv začíná u dráhy života objektu *ControllerClock*, u kterého je použit stereotyp *TimerResource*. Tento objekt bude ve výsledném kódu reprezentován funkční jednotkou, protože periodicky vyvolává operaci *Report* objektu *Reporter*. Tato funkční jednotka bude obsahovat pouze jeden proces, umisťující data a přijímající data z jednoho kanálu, který bude jako vstupní parametr využit u procesu *Report* funkční jednotky reprezentující objekt *Reporter*.

Uvnitř tohoto procesu budou získány data z pole, které bude reprezentovat objekt *Servos_Data*, obsahující data o servech. Toto pole bude vygenerováno do pomocného



Obrázek 9: Sekvenční diagram

kódu. Jeho třída je v *UML* modelu označena stereotypem *PpUnit*, jedná se tedy o pasivní část systému. Proto také nebude během výběru funkčních jednotek vybrána.

Na výstupu tohoto procesu budou umístěna data, reprezentovaná objektem *Status*. Jelikož není v modelu uvedena jeho přesná specifikace, bude zapotřebí tento výstup upravit po vygenerování kódu. Pro zjednodušení může být reprezentován hodnotou *Integer*.

Tato hodnota je pomocí komunikačního kanálu předána procesu *SendStatus*, umístěného ve funkční jednotce *ControllerCommunication*, reprezentující zde objekt *Controller_Comm*.

Objekt *CAN_bus* zde bude přeskočen, jedná se pouze o reprezentanta sběrnice, po které jsou přenášena data. Zde bude stačit pouze předání hodnoty procesu *Transmitted* funkční jednotky *Station_Comm*. Výstup tohoto procesu bude propojen se vstupem procesu *UpdateDisplay* jednotky *DisplayRefresher*. V tomto procesu dojde ke zobrazení přijatých dat na uživatelský výstup.

Objekt *DisplayData* je zde pasivní částí systému, nebude tedy vybrán jako funkční jednotka a bude přeskočen.

12.2.2 Transformace

Postup transformace bude proveden klasickým způsobem s přihlédnutím na výše zmíněné okolnosti, hlavně co se týče provázání jednotlivých procesů komunikačními kanály.

Ukázka výsledného kódu je vidět níže. Celý vygenerovaný kód je umístěn v příloze ve složce *Tool_Workspace*.

```
import "Prelude.pfl"
import "DefaultDiagram.pfl"
import "Deployment_diagram_of_myUMLModel.pfl"

-------- DEVICES --------
```

```
−−−−−−− DEVICE −−−−−−−
UpdateDisplay :: AD Integer −> () −− vlozeni uzivatelskeho vystupu
UpdateDisplay AD = ()
    ...
device_0 :: Device
device_0 = Fair [Process UpdateDisplay, Process UpdateGraphics, Process displayData]

−−−−−−− COMPONENTS of device_0 −−−−−−−
DisplayRefresher = ESComponent "device_0" Emulator
...
−−−−−−− DEVICE −−−−−−−
Invoke :: AA Integer −> (AA Integer)
Invoke AA = (AA)
device_3 :: Device
device_3 = Process Invoke

−−−−−−− COMPONENTS of device_3 −−−−−−−
ControllerClock = ESComponent "device_3" Emulator
    ...
−−−−−−− DEVICE −−−−−−−
Report :: AA Integer −> (AB Integer)
Report AA = (AB)
device_4 :: Device
device_4 = Process Report

−−−−−−− COMPONENTS of device_4 −−−−−−−
Reporter = ESComponent "device_4" Emulator
```

Výpis 16: Vygenerovaný kód systému s robotickým ramenem

Ve výsledném kódu lze například vidět proces *UpdateDisplay*, který má na starost uživatelský výstup. Vypíše přijatou hodnotu *Integer* na obrazovku. Pod ním je v ukázce umístěna definice procesu Invoke funkční jednotky *device_3*, starající se o periodické vyvolání zaslání reportu. Stejný komunikační kanál (AA) s tímto procesem sdílí proces *Report*, jehož definici lze vidět o několik níže.

13 Možnosti rozšíření

Generování kódu z poskytnutého modelu, vytvořeného pomocí jazyka *UML* rozsáhlou problematiku a je zde mnoho různých možností, jak tento nástroj rozšířit. Jednou z možností rozšíření by mohlo být zakomponování možností kompilace a spuštění kódu a jeho následnou analýzu přímo do nástroje.

Dalo by se zde také dopracovat širší využití diagramů chování jazyka *UML* při generování kostry chování a případně zakomponovat informace, obsažené ve stereotypech profilu *MARTE*.

Z pohledu uživatele by se zde určitě více hodilo kvůli větší přehlednosti nějaké vizuální zpracování provázání parametrů jednotlivých procesů. Co se týče výsledných úprav a dokončování kódu uživatelem, bylo by možné například zvýraznit syntaxi vygenerovaného kódu v editoru.

Zjednodušit samotný proces transformace by šlo například i omezením dostupných operací a vlastností klasifikátorů, zobrazených uživateli v okně během provázání procesů. Zobrazeny by zde mohly být pouze prvky, které jsou pro vybranou komponentu v rámci *UML* modelu viditelné.

Zbývá zde také dopracovat využití profilu *MARTE* během získávání informací od uživatele (mohly by být užitečné pro orientaci uživatele během transformace) a jejich následné využití během generování kódu.

Stereotypy profilu *MARTE* by zde mohly být také užitečné například při přechodu od modelu ke skutečnému nasazení vestavného systému. Při analýze modelu a definování struktury samotného systému by mohli být pomocí použitých stereotypů (a definovaného jazyka, ve kterém bude generován kód vestavného systému) vybráni a uživateli nabídnuti vhodní reprezentanti jednotlivých částí systému (dostupné mikroprocesory, paměti atd.)

V tomto směru by se mohlo hodit rozšíření jazyka *e-PFL* o možnost realističtější simulace vestavného systému (například o možnosti úpravy rychlosti zpracování či přenosové rychlosti komunikačních kanálů ve formě anotací a umožňující tak během simulace analyzovat například vytížení jednotlivých funkčních jednotek).

Zajímavou možností by mohla být také tvorba rozšiřujícího modulu pro některý z dostupných modelovacích nástrojů, podporujících profil *MARTE*. Tento plugin by mohl umožňovat generování kódu *e-PFL*, jeho editaci a následné provádění simulace vestavných systémů přímo v daném nástroji.

13.1 UML profil pro jazyk e-PFL

Užitečná by mohla být také tvorba *UML* profilu, šitého na míru jazyku *e-PFL*. Kvůli velké obecnosti extrakce možných zařízení a komponent (viz 10.1.1) a na druhou stranu širokému spektru stereotypů, dostupných v profilu *MARTE*, zaměřených spíše na pozdější fáze analýzy by rychlá úprava *UML* diagramů mohla pomoci uživatelům, kteří nejsou s profilem *MARTE* seznámeni.

Samotný profil by mohl vypadat následovně:

Stereotyp Device - S vlastností *IsFair* a polem typu *EmbSystem* - Název vlastnosti *IsFair* zde hovoří sám za sebe, pole typu *EmbSystem* by zde reprezentoval jednotlivé komponenty, využívající procesy daného zařízení. Tento stereotyp by rozšiřoval klasifikátory (typy, dědící z typu *Classifier*, jako např. třída či rozhraní).

Typ EmbSystem - Typ reprezentující komponentu, obsahující vlastnosti jako *Name*, *Mediator*, a pole anotací.

Typ Rename - Reprezentace anotace *rename*, se dvěma vlastnostmi, reprezentujícími starý a nový název komunikačního kanálu.

Výčet Mediator - Výčet, umožňující definovat, zda bude provedena pouze simulace (literál *Emulator*) či definovat název jazyka, ve kterém bude generován kód reprezentovaného vestavného systému (například literál *MicroNET* či *Hume*).

Stereotyp Process - Tento stereotyp by sloužil pro označení procesu zařízení. Byla by zde obsažena vlastnost *IsPropagated*, která by označovala, zda má být daná vlastnost reprezentující proces použita i v potomcích klasifikátoru, ve kterém je tato vlastnost definována. Tento stereotyp by rozšiřoval prvky metamodelu jazyka *UML*, které dědí z typu *Feature*.

Tento profil by mohl podstatně zjednodušit postup transformace. Případně by se mohlo jednat o rozšíření profilu *MARTE*.

14 Závěr

Tato práce mi umožnila podívat se hlouběji na samotný jazyk *UML* a dovolila mi podívat se také hlouběji na problematiku modelování vestavných systémů. Na práci s vestavnými systémy jsem během svého studia moc často nenarazil, stejně tak na práci s funkcionálními jazyky. Seznámil jsem se zde také s *UML* profilem *MARTE* a obecně s možnostmi tvorby a využití profilů jazyka *UML* při modelování specifických problémů.

Nejvíce času mi zde zabrala implementace knihovny, určené pro načítání *UML* modelu a knihovny reprezentující profil *MARTE*. Knihovnu, určenou pro načítání *UML* modelu jsem kvůli rozsáhlosti metamodelu jazyka *UML* zjednodušil a načítaná data jsem více přizpůsobil struktuře *XMI* souborů.

Při samotné transformaci jsem se zaměřil hlavně na stránku struktury výsledného kódu, umožňující vytvořit základní scénář transformace. Postup generování chování modelovaného vestavného systému jsem rozpracoval v textu práce. Proces transformace *UML* modelu na kód jazyka *e-PFL* lze díky rozsáhlosti jazyka *UML* a profilu *MARTE* ještě více rozšiřovat a je zde mnoho dalších možností, jak tento nástroj rozšířit.

15 Reference

[1] Ing. Marek Běhálek, Ph.D., *Autoreferát disertační práce*, 2011.

[2] Eric Etheridge, *Haskell Tutorial for C Programmers*, 2005.

[3] Object Management Group, *OMG Unified Modeling Language ™ (OMG UML), Superstructure (Version 2.2)*, 2009.

[4] Object Management Group, *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, (Version 1.0)*, 2009.

[5] Object Management Group, *OMG MOF 2 XMI Mapping Specification, Version (2.4.1)*, 2011.

[6] Stephen J. Mellor, *Embedded Systems in UML*, 2007.

[7] Rong Chen, Marco Sgroi, Grant Martin, Luciano Lavagno, Alberto Sangiovanni Vincentelli, Jan Rabaey, *Embedded System Design Using UML and Platforms*